Table of Contents


```
clc; clear all; close all;

rng(1000)
X = gen_truce_track();

T = 0.1;
x0 = zeros(5,1);
P0 = diag([100 100 100 (5*pi/180)^2 (pi/180)^2]);
sensor_placement = kron(ones(1, size(X,2)), [280; -140]);
R = diag([15^2 (2*pi/180)^2]);
Q = diag([0 0 0 0 pi/180*0.001]);
h = @(x,s) rangeBearingMeasurements(x,s);
f = @(x, T) coordinatedTurnMotion(x, T);
spf = @(x, P, type) sigmaPoints(x, P, type);
type = 'CKF';

% a
Y = genNonLinearMeasurementSequence(X, sensor_placement, h, R);
xy = pos_from_meas(Y(2,:), Y(1,:), sensor_placement(:, 2:end));
[xs, Ps, xf, Pf, xp, Pp] = nonLinRTSsmoother(Y, x0, P0, f, T, Q,
 sensor_placement, h, R, spf, type);



plot(X(1,:), X(2,:));
hold on
plot(xs(1,:), xs(2,:));
plot(xf(1,:), xf(2,:));
plot(xy(1,:), xy(2,:), 'o');
plot_error_cov(xs, Ps, 3, 5, '--g')
plot_error_cov(xf, Pf, 3, 5, '--black')

legend('x', 'x_s', 'x_f', 'y', '3\sigma_s', '3\sigma_f')
%legend('x', 'x_s', 'x_f', '3\sigma_s', '3\sigma_f')
title('a')

% b
```

```
[y, H] = rangeBearingMeasurements(X(:, 151) + 100*ones(size(X(:,
 151))), sensor_placement(:,1));
Y(:, 150) = y;
xy = pos_from_meas(Y(2,:), Y(1,:), sensor_placement(:, 2:end));
[xs, Ps, xf, Pf, xp, Pp] = nonLinRTSsmoother(Y, x0, P0, f, T, Q,
 sensor_placement, h, R, spf, type);

figure
plot(X(1,:), X(2,:));
hold on
plot(xs(1,:), xs(2,:));
plot(xf(1,:), xf(2,:));
plot(xy(1,:), xy(2,:), 'o');
plot_error_cov(xs, Ps, 3, 5, '--g')
plot_error_cov(xf, Pf, 3, 5, '--black')

legend('x', 'x_s', 'x_f', 'y', '3\sigma_s', '3\sigma_f')
%legend('x', 'x_s', 'x_f', '3\sigma_s', '3\sigma_f')
title('b')
```

2

```
clc; clear all; close all;
rng(0)

A = 1;
H = 1;
Q = 1.5;
R = 2.5;
P0 = 6;
x0 = 2;
N = 50;
npoints = 50;

X = x0 + mvnrnd(0, P0);
for i = 1:N
    X = [X mvnrnd(X(end), Q)];
end
Y = X(:,2:end) + mvnrnd(0, R, N).';

[Xk, Pk] = kalmanFilter(Y, x0, P0, A, Q, H, R);
```

a

```
close all; clc;
plotFunc_handle_1  = @(k, X, Xmin1, W, j) plotPostPdf(k, X, W, Xk, Pk,
 true, 0.1, [0 100 0 1]);
plotFunc_handle_2  = @(k, X, Xmin1, W, j) plotPostPdf(k, X, W, Xk, Pk,
 false, 0.1, [0 100 0 1]);

%[xfpr, Pfpr, Xpr, Wpr] = pfFilter(x0, P0, Y, @(x) x, Q, @(x)x, R,
 npoints, true, plotFunc_handle_1, false);
```

```
[xfpnr, Pfpnr, Xpnr, Wpnr] = pfFilter(x0, P0, Y, @(x) x, Q, @(x)x, R,
 npoints, false, plotFunc_handle_2, false);
X = X(2:end);


plot(X)
hold on
plot(Xk)
plot(xfpr)
plot(xfpnr)
plot(Y)
legend('x', 'x_{kalman}', 'x_{particle filter with
 resampling}', 'x_{particle filter no resampling}', 'y')

figure
plot(Pk(:))
hold on
plot(Pfpr(:))
plot(Pfpnr(:))
legend('\Sigma_{Kalman}', '\Sigma_{particle filter with
 resampling}', '\Sigma_{particle filter without resampling}')


mse_resampling =  1/N*sum((X-xfpr).^2)
mse_no_resampling =  1/N*sum((X-xfpnr).^2)
```

2b

```
clc; close all;
plotFunc_handle  = @(k, X, Xmin1, W, j)  plotPartTrajs(k, X, Xmin1,
 j);
[xfpnr, Pfpnr, Xpnr, Wpnr] = pfFilter(x0, P0, Y, @(x) x, Q, @(x)x, R,
 npoints, false, plotFunc_handle, false);
hold on
plot(X(2:end), '-oblack')
```

2c

```
clc; close all;
plotFunc_handle  = @(k, X, Xmin1, W, j)  plotPartTrajs(k, X, Xmin1,
 j);
[xfpnr, Pfpnr, Xpnr, Wpnr] = pfFilter(x0, P0, Y, @(x) x, Q, @(x)x, R,
 npoints, true, plotFunc_handle, false);
hold on
plot(X(2:end), '-oblack')
```

3

```
clc; clear; %close all;
rng(0)

% a
```

```
MapProblemGetPoint();

% b
Xv = []; %diff(Xk)
for i = 2:size(Xk,2)
    Xv = [Xv Xk(:,i)-Xk(:,i-1)];
end

R = diag( [0.0005 0.0005 ] );

Q = [0.001 0 0 0
     0 0.001 0 0
     0 0 0.02 0
     0 0 0 0.02];

Y = Xv + mvnrnd(zeros(size(Xk,1), 1), R, size(Xk,2)-1).';


% c
% The probability of being in any of the houses or outside of the map
 is
% 0. p(X_in_house) = p(X_outside_of_map) = 0
```

3d

```
Xk = [Xk(:,1:end-1); Xv];

x0 = Xk(:,1);
P0 = zeros(4);

f = @(x) [x(1,:) + x(3,:)
          x(2,:) + x(4,:)
          x(3,:)
          x(4,:)];

h = @(x) [x(3) + (isOnRoad(x(1), x(2))-1)*10^10; x(4) +
 (isOnRoad(x(1), x(2))-1)*10^10];

npoints = 2000;

plot_xy  = @(k, X, Xmin1, W, j)  plot(X(1,:)*W.', X(2,:)*W.', '-bo');
[xfpnr, Pfpnr, Xpnr, Wpnr] = pfFilter(x0, P0, Y, f, Q, h, R, npoints,
 true, plot_xy, true);
plot(xfpnr(1,:), xfpnr(2,:), '-o')

meas = cumsum(Y.').' + x0(1:2);
%plot(meas(1,:), meas(2,:), '-*');
axis([min([meas(1,:) 0 xfpnr(1,:)]) max([meas(1,:) 12 xfpnr(1,:)])
 min([meas(2,:) 0 xfpnr(2,:)]) max([meas(2,:) 10 xfpnr(2,:)])])

%plot(Xk(1,:), Xk(2,:), 'black')
true_x_cum = cumsum(Xk(3:4,:).').' + x0(1:2);
%plot(true_x_cum(1,:), true_x_cum(2,:), '-black*');
```

```
fprintf("done\n");
fprintf("sum of all filter error = %f\n", sum(sum((abs(Xk-
xfpnr)).').'));
```

3 e

```
Xk = [Xk(:,1:end-1); Xv];

x_0 = Xk(:,1);
P0 = zeros(4);

f = @(x) [x(1,:) + x(3,:)
          x(2,:) + x(4,:)
          x(3,:)
          x(4,:)];

h = @(x) [x(3) + (isOnRoad(x(1), x(2))-1)*10^10; x(4) +
 (isOnRoad(x(1), x(2))-1)*10^10];



npoints = 10000;

dx = sqrt(8*10/npoints);
dy = dx;
x0 = zeros(4, size(1:dy:9, 2)*size(1:dx:11, 2) );
index = 0;
for y = 1:dy:9
    for x = 1:dx:11
        if isOnRoad(x,y)
            index = index + 1;
            x0(:,index) = [x;y;0;0];
        end
    end
end

x0 = x0(:,1:index);
%plot(x0(1,:), x0(2,:), 'o')


R = diag( [0.0005 0.0005 ] );
Q = diag([0.01 0.01 0.001 0.001]);

plot_xy  = @(k, X, Xmin1, W, j)  plot(X(1,:)*W.', X(2,:)*W.', '-bo');
[xfpnr, Pfpnr, Xpnr, Wpnr] = pfFilter(x_0, P0, Y, f, Q, h, R, size(x0,
 2), true, plot_xy, true, x0);
plot(xfpnr(1,:), xfpnr(2,:), '-o')

meas = cumsum(Y.').' + x_0(1:2);
%plot(meas(1,:), meas(2,:), '-*');
axis([min([meas(1,:) 0 xfpnr(1,:)]) max([meas(1,:) 12 xfpnr(1,:)])
 min([meas(2,:) 0 xfpnr(2,:)]) max([meas(2,:) 10 xfpnr(2,:)])])
```

```
%plot(Xk(1,:), Xk(2,:), 'black')
true_x_cum = cumsum(Xk(3:4,:).').' + x_0(1:2);
%plot(true_x_cum(1,:), true_x_cum(2,:), '-black*');

fprintf("done\n");
fprintf("sum of all filter error = %f\n", sum(sum((abs(Xk-
xfpnr)).').'));

%plot(x0(1, :), x0(2,:),'o')
```

new functions

```
function [xfp, Pfp, Xp, Wp] = pfFilter(x_0, P_0, Y, proc_f, proc_Q,
 meas_h, meas_R, N, bResample, plotFunc, three, x0)
%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 = [];
xfp = zeros(n,K);
Pfp = zeros(n,n,K);
Xp = zeros(n,N,K);
```

```matlab
Wp = zeros(N,K);
j = [];

if nargin < 12
    % Draw the first particles for k=0
    X = x_0 + mvnrnd(zeros(size(x_0)), P_0, N).';
else
    X = x0;
end
W = ones(1, size(X,2)) / N;

% 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, three);

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



    % Update the outputs
    Wp(:,i) = W.';
    Xp(:,:,i) = X(:,1:N);
    xfp(:,i) = 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, three)
%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)
    y = meas_h(X_k(:, i));
    W_k(i) = W_kmin1(i) * mvnpdf(yk, y, 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
        try
            % 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
        catch
            printf("You have big problems, boy.");
        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
```

old function

```
function plotPostPdf(k, Xk, Wk, xf, Pf, bResample, sigma, ax)
%PLOTPOSTPDF Plots blurred pdf for a PF posterior, and plots a Kalman
% posterior to compare with.
%
%   This function is intended to be used as a function handle for a
%   compatible particle filter function. It is meant to be called each
 time
%   the particle filter has updated the particles (but before any
%   resampling has been carried out.)
%
%   To use it in your filter you should first compute xf, Pf, and set
%   bResample, sigma and ax.
%   Then define a function handle
%       plotFunc_handle  = @(k, Xk, Xkmin1, Wk, j) ...
%                          (plotPostPdf(k, Xk, Wk, xf, Pf, bResample,
 sigma, ax))
%   Then call your PF-function with plotFunc_handle as plotFunc
 argument.
%
% Inputs:
%   k           time instance index
%   Xk          [n x N] N particles of dimension n to approximate
 p(x_k).
%   Wk          [1 x N] Corresponding weights.
%   xf          [n x K] Filter posteriors for some filter to compare
 with
%   Pf          [n x n x K] Filter posterior covariances for ^
%   bResample   Flag for resampling. False: do not resample, True:
 resample
%   sigma       Controls the kernel width for blurring particles.
%   ax          [xmin xmax ymin ymax] Used for setting the x-axis
 limits to
%               a value that doesn't change through iterations of the
 PF
%               filter.

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

```
    % Let us first determine the x-interval of interest:
    xmin =    min(Xk(1,:)); %ax(1);
    xmax =    max(Xk(1,:)); %ax(2);
    X    =    linspace(xmin-(xmax-xmin)/3, xmax+(xmax-xmin)/3, 800);

    % We can now construct a continuous approximation to the posterior
    % density by placing a Gaussian kernel around each particle
    pApprox = zeros(size(X));   % A vector that will contain the pdf
 values

    if bResample
        sigma=(xmax-xmin)/sqrt(N);
    end

    for i = 1 : N
        pApprox = pApprox...
            + Wk(1,i)...
            *normpdf(Xk(1,i), X, sigma);
    end

    % We are now ready to plot the densities

    % figure;
    set(gcf, 'Name', ['p_',num2str(k), '_', 'SIR']);
    % clf

    plot(X, pApprox, 'LineWidth', 2)   % This is the PF approximation
    hold on
    plot(X, normpdf(xf(1,k), X, sqrt(Pf(1,1,k))), 'r-.', 'LineWidth',
 2) % KF posterior density
    legend('Particle filter approximation', 'Kalman
 filter', 'Location', 'southwest')
    title(['p(x_k |  y_{1:k}), k=', num2str(k)])
    hold off;
    pause()
end

function plotPartTrajs(k, Xk, Xkmin1,  j)
%PLOTPARTTRAJS Summary of this function goes here
%   Plots lines between ith sample of Xk and j(i)th sample of Xk-1.
 When
%   repeated during a particle filter execution, this will produce
 particle
%   trajectories illustration over time.
%
%   This function is intended to be passed as a function handle into
 your
%   particle filter function.
%
% Inputs:
%   k           time instance index
%   Xk          [n x N] N particles of dimension n to approximate
 p(x_k).
```

```
%   Xkmin1      [n x N] N particles of dimension n to approximate
 p(x_k-1).
%   Wk          [1 x N] Corresponding weights.
%   j           Index vector such that Xk(:,i) = Xkmin1(:,j(i))

    if (size(Xk,2) <= 50) % At most 50 particles may be plotted
        for i = 1:size(Xk,2) % loop through all particles
            plot([k-1 k], [Xkmin1(1,j(i)) Xk(1,i)]);
            hold on
        end
        title(['Particle trajectories up to time k=', num2str(k)]);
        pause(0.05);
    else
        disp('Too many particles to plot!');
    end
end

function a = plot_pdf_pf(X, W, color)
    W = W.';
    [unique_x, ind] = unique(X.', 'rows');
    sum_w = zeros(1, size(ind,1));

    for i = 1:size(ind,1)
        cur_x = X(:, ind(i));
        for j = 1 : size(X,2)
           sum_w(i) = sum_w(i) + W(j)*(cur_x == X(:,j));
        end
    end

    sum_w = sum_w / (trapz(unique_x, sum_w)); % Cheating?
    if nargin < 3
        plot(unique_x, sum_w, 'o');
    else
       plot(unique_x, sum_w, color);
    end

end

function [X, P] = kalmanFilter(Y, x_0, P_0, A, Q, H, R)
%KALMANFILTER Filters measurements sequense Y using a Kalman filter.
%
%Input:
%   Y           [m x N] Measurement sequence
%   x_0         [n x 1] Prior mean
%   P_0         [n x n] Prior covariance
%   A           [n x n] State transition matrix
%   Q           [n x n] Process noise covariance
%   H           [n x n] Measruement model matrix
%   R           [n x n] Measurement noise covariance
%
%Output:
%   x           [n x N] Estimated state vector sequence
%   P           [n x n x N] Filter error convariance
%
```

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

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

    % Data allocation
    X = zeros(n,N);
    P = zeros(n,n,N);

    % Filter
    for k = 1:N

        if k == 1 % Initiate filter

            % Time prediction
            [xPred, PPred] = linearPrediction(x_0, P_0, A, Q);

        else

            % Time prediction
            [xPred, PPred] = linearPrediction(X(:,k-1), P(:,:,k-1), A,
 Q);

        end

        % Measurement update
        [X(:,k), P(:,:,k)] = linearUpdate(xPred, PPred, Y(:,k), H, R);

    end
end

function [x, P] = linearPrediction(x, P, A, Q)
%LINEARPREDICTION calculates mean and covariance of predicted state
%   density using a liear Gaussian model.
%
%Input:
%   x           [n x 1] Prior mean
%   P           [n x n] Prior covariance
%   A           [n x n] State transition matrix
%   Q           [n x n] Process noise covariance
%
%Output:
%   x           [n x 1] predicted state mean
%   P           [n x n] predicted state covariance
%

% Predicted mean
x = A*x;

% Predicted Covariance
P = A*P*A' + Q;
```

```
end

function [x, P] = linearUpdate(x, P, y, H, R)
%LINEARPREDICTION calculates mean and covariance of predicted state
%   density using a liear Gaussian model.
%
%Input:
%   x           [n x 1] Prior mean
%   P           [n x n] Prior covariance
%   H           [n x n] Measruement model matrix
%   R           [n x n] Measurement noise covariance
%
%Output:
%   x           [n x 1] updated state mean
%   P           [n x n] updated state covariance
%

% Innovation
v = y - H*x;
S = H*P*H' + R;

% Kalman gain
K = P*H'/S;

% Updated mean and covariance
x = x + K*v;
P = P - K*S*K';

end

function a = plot_error_cov(mu, cov, level, every, colour)
    for i = 1:every:size(mu,2)
        [xy] = sigmaEllipse2D(mu(1:2,i), cov(1:2,1:2,i), level, 256);
        plot(xy(1,:), xy(2,:), colour);
    end
end

function [ xy ] = sigmaEllipse2D( mu, Sigma, level, npoints )
%SIGMAELLIPSE2D generates x,y-points which lie on the ellipse
 describing
% a sigma level in the Gaussian density defined by mean and
 covariance.
%
%Input:
%   MU          [2 x 1] Mean of the Gaussian density
%   SIGMA       [2 x 2] Covariance matrix of the Gaussian density
%   LEVEL       Which sigma level curve to plot. Can take any positive
 value,
%               but common choices are 1, 2 or 3. Default = 3.
%   NPOINTS     Number of points on the ellipse to generate. Default =
 32.
%
%Output:
```

```
%   XY          [2 x npoints] matrix. First row holds x-coordinates,
 second
%               row holds the y-coordinates. First and last columns
 should
%               be the same point, to create a closed curve.


%Setting default values, in case only mu and Sigma are specified.
if nargin < 3
    level = 3;
end
if nargin < 4
    npoints = 32;
end

% Create a vector of angles. The angles are those to create the level
 curve of the distribution.
% The vector starts at 0 and ends at 0 in order to creat a full
 elipse. In between the zeros there are npoints-2 points.
fi = [0:2*pi/(npoints-1):2*pi-2*pi/(npoints-1) 0];

xy = level*sqrtm(Sigma)*[cos(fi); sin(fi)] + mu;

end

function X = gen_truce_track()
```

True track

Sampling period

```
    T = 0.1;
    % Length of time sequence
    K = 600;
    % Allocate memory
    omega = zeros(1,K+1);
    % Turn rate
    omega(200:400) = -pi/201/T;
    % Initial state
    x0 = [0 0 20 0 omega(1)]';
    % Allocate memory
    X = zeros(length(x0),K+1);
    X(:,1) = x0;
    % Create true track
    for i=2:K+1
        % Simulate
        X(:,i) = coordinatedTurnMotion(X(:,i-1), T);
        % Set turn-rate
        X(5,i) = omega(i);
    end

end

function [h, H] = rangeBearingMeasurements(x, s)
```

```
%RANGEBEARINGMEASUREMENTS calculates the range and the bearing to the
%position given by the state vector x, from a sensor locateed in s
%
%Input:
%   x           [n x 1] State vector
%   s           [2 x 1] Sensor position
%
%Output:
%   h           [2 x 1] measurement vector
%   H           [2 x n] measurement model Jacobian
%
% NOTE: the measurement model assumes that in the state vector x, the
 first
% two states are X-position and Y-position.

    % Range
    rng = norm(x(1:2)-s);
    % Bearing
    ber = atan2(x(2)-s(2),x(1)-s(1));
    % Measurement vector
    h = [rng;ber];

    % Measurement model Jacobian
    H = [(x(1)-s(1))/rng      (x(2)-s(2))/rng     0 0 0;
        -(x(2)-s(2))/(rng^2) (x(1)-s(1))/(rng^2) 0 0 0];

end

function [xy] = pos_from_meas(ber, rng, sensor_placement)
    xy = [rng.*cos(ber)
          rng.*sin(ber)];
    xy = xy + sensor_placement;
end

function [f, F] = coordinatedTurnMotion(x, T)
%COORDINATEDTURNMOTION calculates the predicted state using a
 coordinated
%turn motion model, and also calculated the motion model Jacobian
%
%Input:
%   x           [5 x 1] state vector
%   T           [1 x 1] Sampling time
%
%Output:
%   f           [5 x 1] predicted state
%   F           [5 x 5] motion model Jacobian
%
% NOTE: the motion model assumes that the state vector x consist of
 the
% following states:
%   px          X-position
%   py          Y-position
%   v           velocity
%   phi         heading
```

```
%   omega       turn-rate

    % Velocity
    v = x(3);
    % Heading
    phi = x(4);
    % Turn-rate
    omega = x(5);

    % Predicted state
    f = x + [
        T*v*cos(phi);
        T*v*sin(phi);
        0;
        T*omega;
        0];

    % Motion model Jacobian
    F = [
        1 0 T*cos(phi) -T*v*sin(phi) 0;
        0 1 T*sin(phi) T*v*cos(phi)  0;
        0 0 1          0             0;
        0 0 0          1             T;
        0 0 0          0             1
        ];
end

function X = genNonLinearStateSequence(x_0, P_0, f, T, Q, N)
%GENLINEARSTATESEQUENCE generates an N-long sequence of states using a
%    Gaussian prior and a linear Gaussian process model
%
%Input:
%   x_0         [n x 1] Prior mean
%   P_0         [n x n] Prior covariance
%   f           Motion model function handle
%   T           Sampling time
%   Q           [n x n] Process noise covariance
%   N           [1 x 1] Number of states to generate
%
%Output:
%   X           [n x N] State vector sequence
%

    % Dimension of state vector
    n = length(x_0);

    % allocate memory
    X = zeros(n, N);

    % Generete start state
    X(:,1) = mvnrnd(x_0', P_0)';

    % Generate sequence
    for k = 2:N+1
```

```
        % generate noise vector
        q = mvnrnd(zeros(1,n), Q)';

        % Propagate through process model
        [fX, ~] = f(X(:,k-1),T);
        X(:,k) = fX + q;

    end

end

function Y = genNonLinearMeasurementSequence(X, S, h, R)
%GENNONLINEARMEASUREMENTSEQUENCE generates ovservations of the states
% sequence X using a non-linear measurement model.
%
%Input:
%   X           [n x N+1] State vector sequence
%   S           [n x N] Sensor position vector sequence
%   h           Measurement model function handle
%   R           [m x m] Measurement noise covariance
%
%Output:
%   Y           [m x N] Measurement sequence
%

    % Parameters
    N = size(X,2);
    m = size(R,1);

    % Allocate memory
    Y = zeros(m,N-1);

    for k = 1:N-1
        % Measurement
        [hX,~] = h(X(:,k+1),S(:,k));
        % Add noise
        Y(:,k) = hX + mvnrnd(zeros(1,m), R)';

    end

end

function [SP,W] = sigmaPoints(x, P, type)
% SIGMAPOINTS computes sigma points, either using unscented transform
 or
% using cubature.
%
%Input:
%   x           [n x 1] Prior mean
%   P           [n x n] Prior covariance
%
%Output:
%   SP          [n x 2n+1] matrix with sigma points
```

```
%   W           [1 x 2n+1] vector with sigma point weights
%

    switch type
        case 'UKF'

            % Dimension of state
            n = length(x);

            % Allocate memory
            SP = zeros(n,2*n+1);

            % Weights
            W = [1-n/3 repmat(1/6,[1 2*n])];

            % Matrix square root
            sqrtP = sqrtm(P);

            % Compute sigma points
            SP(:,1) = x;
            for i = 1:n
                SP(:,i+1) = x + sqrt(1/2/W(i+1))*sqrtP(:,i);
                SP(:,i+1+n) = x - sqrt(1/2/W(i+1+n))*sqrtP(:,i);
            end

        case 'CKF'

            % Dimension of state
            n = length(x);

            % Allocate memory
            SP = zeros(n,2*n);

            % Weights
            W = repmat(1/2/n,[1 2*n]);

            % Matrix square root
            sqrtP = sqrtm(P);

            % Compute sigma points
            for i = 1:n
                SP(:,i) = x + sqrt(n)*sqrtP(:,i);
                SP(:,i+n) = x - sqrt(n)*sqrtP(:,i);
            end

        otherwise
            error('Incorrect type of sigma point')
    end
end

function [x, P] = nonLinKFprediction(x, P, f, T, Q, sigmaPoints, type)
%NONLINKFPREDICTION calculates mean and covariance of predicted state
%   density using a non-linear Gaussian model.
%
```

```
%Input:
%   x           [n x 1] Prior mean
%   P           [n x n] Prior covariance
%   f           Motion model function handle
%   T           Sampling time
%   Q           [n x n] Process noise covariance
%   sigmaPoints Handle to function that generates sigma points.
%   type        String that specifies the type of non-linear filter
%
%Output:
%   x           [n x 1] predicted state mean
%   P           [n x n] predicted state covariance
%

    switch type
        case 'EKF'

            % Evaluate motion model
            [fx, Fx] = f(x,T);
            % State prediction
            x = fx;
            % Covariance prediciton
            P = Fx*P*Fx' + Q;
            % Make sure P is symmetric
            P = 0.5*(P + P');

        case 'UKF'

            % Predict
            [x, P] = predictMeanAndCovWithSigmaPoints(x, P, f, T, Q,
 sigmaPoints, type);

            if min(eig(P))<=0
                [v,e] = eig(P);
                emin = 1e-3;
                e = diag(max(diag(e),emin));
                P = v*e*v';
            end

        case 'CKF'

            % Predict
            [x, P] = predictMeanAndCovWithSigmaPoints(x, P, f, T, Q,
 sigmaPoints, type);

        otherwise
            error('Incorrect type of non-linear Kalman filter')
    end
end

function [x, P] = nonLinKFupdate(x, P, y, s, h, R, sigmaPoints, type)
%NONLINKFUPDATE calculates mean and covariance of predicted state
%   density using a non-linear Gaussian model.
%
```

```
%Input:
%   x           [n x 1] Prior mean
%   P           [n x n] Prior covariance
%   y           [m x 1] measurement vector
%   s           [2 x 1] sensor position vector
%   h           Measurement model function handle
%   R           [n x n] Measurement noise covariance
%   sigmaPoints Handle to function that generates sigma points.
%   type        String that specifies the type of non-linear filter
%
%Output:
%   x           [n x 1] updated state mean
%   P           [n x n] updated state covariance
%


switch type
    case 'EKF'

        % Evaluate measurement model
        [hx, Hx] = h(x,s);

        % Innovation covariance
        S = Hx*P*Hx' + R;
        % Kalman gain
        K = (P*Hx')/S;

        % State update
        x = x + K*(y - hx);
        % Covariance update
        P = P - K*S*K';

        % Make sure P is symmetric
        P = 0.5*(P + P');

    case 'UKF'

        % Update mean and covariance
        [x, P] = updateMeanAndCovWithSigmaPoints(x, P, y, s, h, R,
 sigmaPoints, type);

        if min(eig(P))<=0
            [v,e] = eig(P);
            emin = 1e-3;
            e = diag(max(diag(e),emin));
            P = v*e*v';
        end

    case 'CKF'

        % Update mean and covariance
        [x, P] = updateMeanAndCovWithSigmaPoints(x, P, y, s, h, R,
 sigmaPoints, type);
```

```
    otherwise
        error('Incorrect type of non-linear Kalman filter')
end

end

function [x, P] = predictMeanAndCovWithSigmaPoints(x, P, f, T, Q,
 sigmaPoints, type)
%
%PREDICTMEANANDCOVWITHSIGMAPOINTS computes the predicted mean and
 covariance
%
%Input:
%   x           [n x 1] mean vector
%   P           [n x n] covariance matrix
%   f           measurement model function handle
%   T           sample time
%   Q           [m x m] process noise covariance matrix
%
%Output:
%   x           [n x 1] Updated mean
%   P           [n x n] Updated covariance
%

    % Compute sigma points
    [SP,W] = sigmaPoints(x, P, type);

    % Dimension of state and number of sigma points
    [n, N] = size(SP);

    % Allocate memory
    fSP = zeros(n,N);

    % Predict sigma points
    for i = 1:N
        [fSP(:,i),~] = f(SP(:,i),T);
    end

    % Compute the predicted mean
    x = sum(fSP.*repmat(W,[n, 1]),2);

    % Compute predicted covariance
    P = Q;
    for i = 1:N
        P = P + W(i)*(fSP(:,i)-x)*(fSP(:,i)-x)';
    end

    % Make sure P is symmetric
    P = 0.5*(P + P');

end

function [x, P] = updateMeanAndCovWithSigmaPoints(x, P, y, s, h, R,
 sigmaPoints, type)
```

```
%
%UPDATEGAUSSIANWITHSIGMAPOINTS computes the updated mean and
 covariance
%
%Input:
%   x           [n x 1] Prior mean
%   P           [n x n] Prior covariance
%   y           [m x 1] measurement
%   s           [2 x 1] sensor position
%   h           measurement model function handle
%   R           [m x m] measurement noise covariance matrix
%
%Output:
%   x           [n x 1] Updated mean
%   P           [n x n] Updated covariance
%

    % Compute sigma points
    [SP,W] = sigmaPoints(x, P, type);

    % Dimension of measurement
    m = size(R,1);

    % Dimension of state and number of sigma points
    [n, N] = size(SP);

    % Predicted measurement
    yhat = zeros(m,1);
    hSP = zeros(m,N);
    for i = 1:N
        [hSP(:,i),~] = h(SP(:,i),s);
        yhat = yhat + W(i)*hSP(:,i);
    end

    % Cross covariance and innovation covariance
    Pxy = zeros(n,m);
    S = R;
    for i=1:N
        Pxy = Pxy + W(i)*(SP(:,i)-x)*(hSP(:,i)-yhat)';
        S = S + W(i)*(hSP(:,i)-yhat)*(hSP(:,i)-yhat)';
    end

    % Ensure symmetry
    S = 0.5*(S+S');

    % Updated mean
    x = x+Pxy*(S\(y-yhat));
    P = P - Pxy*(S\(Pxy'));

    % Ensure symmetry
    P = 0.5*(P+P');

end
```

Published with MATLAB® R2020a