Table of Contents

1 1
2 3
3 7
3b 8

1

```
clc; close all; clear;

s1 = [0; 100];
s2 = [100; 0];
R = (0.1*pi/180)^2;

x1_hat = [120; 120];
Sigma_1 = [25 0
           0 100];

x2_hat = [120; -20];
Sigma_2 = [25 0
           0 100];

% Take random samples
n = 10^4;
x1 = mvnrnd(x1_hat, Sigma_1, n).';
x2 = mvnrnd(x2_hat, Sigma_2, n).';

% Measure the samples
[hx1, Hx1] = dualBearingMeasurement(x1, s1, s2);
[hx2, Hx2] = dualBearingMeasurement(x2, s1, s2);

% Add noise to the samples
y1 = [hx1(1,:) + mvnrnd(0, R, n).'
      hx1(2,:) + mvnrnd(0, R, n).'];

y2 = [hx2(1,:) + mvnrnd(0, R, n).'
      hx2(2,:) + mvnrnd(0, R, n).'];

% a
% Calculate the "true" mean and cov
y1_mean_a = (sum(y1.')/n).';
y2_mean_a = (sum(y2.')/n).';
y1_diff = y1-kron(ones(1, n), y1_mean_a);
y2_diff = y2-kron(ones(1, n), y2_mean_a);
S1_a = (y1_diff*y1_diff.')/(n-1);
S2_a = (y2_diff*y2_diff.')/(n-1);

% b
meas_fun = @(x) dualBearingMeasurement(x, s1, s2);
```

```
% EKF
[x, P, S1_b1_ekf, y1_mean_b1_ekf] = nonLinKFupdate(x1_hat, Sigma_1,
 y1(:,1), meas_fun, R, 'EKF');
[x, P, S2_b1_ekf, y2_mean_b1_ekf] = nonLinKFupdate(x2_hat, Sigma_2,
 y2(:,1), meas_fun, R, 'EKF');

% UKF
[x, P, S1_b1_ukf, y1_mean_b1_ukf] = nonLinKFupdate(x1_hat, Sigma_1,
 y1(:,1), meas_fun, R, 'UKF');
[x, P, S2_b1_ukf, y2_mean_b1_ukf] = nonLinKFupdate(x2_hat, Sigma_2,
 y2(:,1), meas_fun, R, 'UKF');

% CKF
[x, P, S1_b1_ckf, y1_mean_b1_ckf] = nonLinKFupdate(x1_hat, Sigma_1,
 y1(:,1), meas_fun, R, 'CKF');
[x, P, S2_b1_ckf, y2_mean_b1_ckf] = nonLinKFupdate(x2_hat, Sigma_2,
 y2(:,1), meas_fun, R, 'CKF');


% c
% P1 first
y1_ekf_level_curve = sigmaEllipse2D(y1_mean_b1_ekf, S1_b1_ekf, 3,
 256);
plot(y1(1,:), y1(2,:) , 'o')
hold on
plot(y1_ekf_level_curve(1,:), y1_ekf_level_curve(2,:), 'black')
plot(y1_mean_b1_ekf(1), y1_mean_b1_ekf(2), '*')


y1_ukf_level_curve = sigmaEllipse2D(y1_mean_b1_ukf, S1_b1_ukf, 3,
 256);
plot(y1_ukf_level_curve(1,:), y1_ukf_level_curve(2,:), 'm')
plot(y1_mean_b1_ukf(1), y1_mean_b1_ukf(2), '*')

y1_ckf_level_curve = sigmaEllipse2D(y1_mean_b1_ckf, S1_b1_ckf, 3,
 256);
plot(y1_ckf_level_curve(1,:), y1_ckf_level_curve(2,:), 'r')
plot(y1_mean_b1_ckf(1), y1_mean_b1_ckf(2), '*')

MC_level_curve = sigmaEllipse2D(y1_mean_a, S1_a, 3, n);
plot(MC_level_curve(1,:), MC_level_curve(2,:), 'g');
plot(y1_mean_a(1), y1_mean_a(2), '*')
legend('y', 'ekf-3\sigma', 'ekf-mean', 'ukf-3\sigma', 'ukf-
mean', 'ckf-3\sigma', 'ckf-mean', 'MC-3\sigma', 'MC-mean')
title('P_1')



% P2 second
figure
y2_ekf_level_curve = sigmaEllipse2D(y2_mean_b1_ekf, S2_b1_ekf, 3,
 256);
plot(y2(1,:), y2(2,:) , 'o')
```

```
hold on
plot(y2_ekf_level_curve(1,:), y2_ekf_level_curve(2,:), 'black')
plot(y2_mean_b1_ekf(1), y2_mean_b1_ekf(2), '*')

y2_ukf_level_curve = sigmaEllipse2D(y2_mean_b1_ukf, S2_b1_ukf, 3,
 256);
plot(y2_ukf_level_curve(1,:), y2_ukf_level_curve(2,:), 'm')
plot(y2_mean_b1_ukf(1), y2_mean_b1_ukf(2), '*')

y2_ckf_level_curve = sigmaEllipse2D(y2_mean_b1_ckf, S2_b1_ckf, 3,
 256);
plot(y2_ckf_level_curve(1,:), y2_ckf_level_curve(2,:), 'r')
plot(y2_mean_b1_ckf(1), y2_mean_b1_ckf(2), '*')

MC_level_curve = sigmaEllipse2D(y2_mean_a, S2_a, 3, n);
plot(MC_level_curve(1,:), MC_level_curve(2,:), 'g');
plot(y2_mean_a(1), y2_mean_a(2), '*')
legend('y', 'ekf-3\sigma', 'ekf-mean', 'ukf-3\sigma', 'ukf-
mean', 'ckf-3\sigma', 'ckf-mean', 'MC-3\sigma', 'MC-mean')
title('P_2')
```

2

```
clc; close all; clear;
% set up parameters
rng(0)
x0 = [0 0 14 0 0].';
P0 = diag([100 100 4 (pi/180)^2 (5*pi/180)^2]);
S1 = [-200; 100];
S2 = [-200; -100];
T = 1;
n = 100;
f = @(x) coordinatedTurnMotion(x, T);
h = @(x) dualBearingMeasurement(x, S1, S2);

R1 = (diag([10*pi/180, 0.5*pi/180])).^2;
R2 = (eye(2) * 0.5*pi/180).^2;

Q1 = zeros(5);
Q1(3,3) = 1;
Q1(5,5) = (pi/180)^2;
Q2 = Q1;


% a
% Generate state, measurement and filter sequences
X = genNonLinearStateSequence(x0, P0, f, Q1, n);
Y = genNonLinearMeasurementSequence(X, h, R1);

[xf_ekf, Pf_ekf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q1, h,
 R1, 'EKF');
[xf_ukf, Pf_ukf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q1, h,
 R1, 'UKF');
```

```
[xf_ckf, Pf_ckf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q1, h,
 R1, 'CKF');

X = X(:, 2:end);


[meas_x,meas_y] = convert_measurement_to_position(Y(1, :), Y(2, :),
 S1, S2);


% PLot the sequences
%EKF
plot(X(1,1:5:end), X(2,1:5:end), 'or')
hold on
plot(xf_ekf(1, 1:5:end), xf_ekf(2, 1:5:end), 'og')
plot(meas_x(1,1:5:end), meas_y(1,1:5:end), 'o')
plot_every_fifth(xf_ekf, Pf_ekf, 3)
legend('x', 'x_f', 'y', '3\sigma')
title('Case 1, ekf')

% CKF
figure
plot(X(1,1:5:end), X(2,1:5:end), 'or')
hold on
plot(xf_ukf(1, 1:5:end), xf_ukf(2, 1:5:end), 'og')
plot(meas_x(1,1:5:end), meas_y(1,1:5:end), 'o')
plot_every_fifth(xf_ukf, Pf_ukf, 3)
legend('x', 'x_f', 'y', '3\sigma')
title('Case 1, ukf')

% UKF
figure
plot(X(1,1:5:end), X(2,1:5:end), 'or')
hold on
plot(xf_ckf(1, 1:5:end), xf_ckf(2, 1:5:end), 'og')
plot(meas_x(1,1:5:end), meas_y(1,1:5:end), 'o')
plot_every_fifth(xf_ckf, Pf_ckf, 3)
legend('x', 'x_f', 'y', '3\sigma')
title('Case 1, ckf')

% b
% Generate state, measurement and filter sequences
X = genNonLinearStateSequence(x0, P0, f, Q2, n);
Y = genNonLinearMeasurementSequence(X, h, R2);

[xf_ekf, Pf_ekf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q2, h,
 R2, 'EKF');
[xf_ukf, Pf_ukf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q2, h,
 R2, 'UKF');
[xf_ckf, Pf_ckf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q2, h,
 R2, 'CKF');

X = X(:, 2:end);
```

```
[meas_x,meas_y] = convert_measurement_to_position(Y(1, :), Y(2, :),
 S1, S2);

% PLot the sequences
%EKF
figure
plot(X(1,1:5:end), X(2,1:5:end), 'or')
hold on
plot(xf_ekf(1, 1:5:end), xf_ekf(2, 1:5:end), 'og')
plot(meas_x(1,1:5:end), meas_y(1,1:5:end), 'o')
plot_every_fifth(xf_ekf, Pf_ekf, 3)
legend('x', 'x_f', 'y', '3\sigma')
title('Case 2, ekf')

% CKF
figure
plot(X(1,1:5:end), X(2,1:5:end), 'or')
hold on
plot(xf_ukf(1, 1:5:end), xf_ukf(2, 1:5:end), 'og')
plot(meas_x(1,1:5:end), meas_y(1,1:5:end), 'o')
plot_every_fifth(xf_ukf, Pf_ukf, 3)
legend('x', 'x_f', 'y', '3\sigma')
title('Case 2, ukf')

% UKF
figure
plot(X(1,1:5:end), X(2,1:5:end), 'or')
hold on
plot(xf_ckf(1, 1:5:end), xf_ckf(2, 1:5:end), 'og')
plot(meas_x(1,1:5:end), meas_y(1,1:5:end), 'o')
plot_every_fifth(xf_ckf, Pf_ckf, 3)
legend('x', 'x_f', 'y', '3\sigma')
title('Case 2 , ckf')

% c
rng(10)
close all; clc
error_ekf = [];
error_ckf = [];
error_ukf = [];
N = 100;

% Simulate N amount of sequences and store the estimation error
for i = 1 : N
    [X, Y, xf_ekf, Pf_ekf, xf_ukf, Pf_ukf, xf_ckf, Pf_ckf] =
 generate_data_and_estimate(x0, P0, f, Q1, R1, n, h);
    error_ekf = [error_ekf X-xf_ekf];
    error_ukf = [error_ekf X-xf_ukf];
    error_ckf = [error_ekf X-xf_ckf];
end

% Plot histograms of the estimation errors
figure
subplot(2,3,1)
```

```
histo = histogram(error_ekf(1,:), 'Normalization', 'pdf');
title('Case 1 - Estimation error ekf - x')
axis([-sqrt(cov(error_ekf(1,:)))*3 sqrt(cov(error_ekf(1,:)))*3 0
 max(histo.Values)])


subplot(2,3,4)
histo = histogram(error_ekf(2,:), 'Normalization', 'pdf');
title('Case 1 - Estimation error ekf - y')
axis([-sqrt(cov(error_ekf(2,:)))*3 sqrt(cov(error_ekf(2,:)))*3 0
 max(histo.Values)])

subplot(2,3,2)
histo = histogram(error_ukf(1,:), 'Normalization', 'pdf');
title('Case 1 - Estimation error ukf - x')
axis([-sqrt(cov(error_ukf(1,:)))*3 sqrt(cov(error_ukf(1,:)))*3 0
 max(histo.Values)])

subplot(2,3,5)
histo = histogram(error_ukf(2,:), 'Normalization', 'pdf');
title('Case 1 - Estimation error ukf - y')
axis([-sqrt(cov(error_ukf(2,:)))*3 sqrt(cov(error_ukf(2,:)))*3 0
 max(histo.Values)])

subplot(2,3,3)
histo = histogram(error_ckf(1,:), 'Normalization', 'pdf');
title('Case 1 - Estimation error ckf - x')
axis([-sqrt(cov(error_ckf(1,:)))*3 sqrt(cov(error_ckf(1,:)))*3 0
 max(histo.Values)])

subplot(2,3,6)
histo = histogram(error_ckf(2,:), 'Normalization', 'pdf');
title('Case 1 - Estimation error ckf - y')
axis([-sqrt(cov(error_ckf(2,:)))*3 sqrt(cov(error_ckf(2,:)))*3 0
 max(histo.Values)])


error_ekf = [];
error_ckf = [];
error_ukf = [];
N = 100;
rng(11)

for i = 1 : N
    [X, Y, xf_ekf, Pf_ekf, xf_ukf, Pf_ukf, xf_ckf, Pf_ckf] =
 generate_data_and_estimate(x0, P0, f, Q2, R2, n, h);
    error_ekf = [error_ekf X-xf_ekf];
    error_ukf = [error_ekf X-xf_ukf];
    error_ckf = [error_ekf X-xf_ckf];
end

figure
subplot(2,3,1)
histo = histogram(error_ekf(1,:), 'Normalization', 'pdf');
```

```
title('Case 2 - Estimation error ekf - x')
axis([-sqrt(cov(error_ekf(1,:)))*3 sqrt(cov(error_ekf(1,:)))*3 0
 max(histo.Values)])

subplot(2,3,4)
histo = histogram(error_ekf(2,:), 'Normalization', 'pdf');
title('Case 2 - Estimation error ekf - y')
axis([-sqrt(cov(error_ekf(2,:)))*3 sqrt(cov(error_ekf(2,:)))*3 0
 max(histo.Values)])

subplot(2,3,2)
histo = histogram(error_ukf(1,:), 'Normalization', 'pdf');
title('Case 2 - Estimation error ukf - x')
axis([-sqrt(cov(error_ukf(1,:)))*3 sqrt(cov(error_ukf(1,:)))*3 0
 max(histo.Values)])

subplot(2,3,5)
histo = histogram(error_ukf(2,:), 'Normalization', 'pdf');
title('Case 2 - Estimation error ukf - y')
axis([-sqrt(cov(error_ukf(2,:)))*3 sqrt(cov(error_ukf(2,:)))*3 0
 max(histo.Values)])

subplot(2,3,3)
histo = histogram(error_ckf(1,:), 'Normalization', 'pdf');
title('Case 2 - Estimation error ckf - x')
axis([-sqrt(cov(error_ckf(1,:)))*3 sqrt(cov(error_ckf(1,:)))*3 0
 max(histo.Values)])

subplot(2,3,6)
histo = histogram(error_ckf(2,:), 'Normalization', 'pdf');
title('Case 2 - Estimation error ckf - y')
axis([-sqrt(cov(error_ckf(2,:)))*3 sqrt(cov(error_ckf(2,:)))*3 0
 max(histo.Values)])
```

3

```
clc; close all; clear;
% Set up simulation parameters
rng(0)
[X, T] = truce_track();

x0 = zeros(5,1);
P0 = diag([100, 100, 100 (5*pi/180)^2 (1*pi/180)^2]);
s1 = [280, -80].';
s2 = [280, -200].';
R = (eye(2) * 4*pi/180).^2;

f = @(x)coordinatedTurnMotion(x, T);
h = @(x)dualBearingMeasurement(x, s1, s2);


% a
% Generate different model noise values
```

```
close all; clc
Q1 = diag([0 0 1 0 pi/180]);
Q2 = diag([0 0 1*10^2 0 pi/180]);
Q3 = diag([0 0 1 0 pi/180*10^2]);
Q4 = diag([0 0 1 0 pi/180])*10^2;
Q5 = diag([0 0 1/100 0 pi/180]);
Q6 = diag([0 0 1 0 pi/180/100]);
Q7 = diag([0 0 1 0 pi/180])/100;
Q = [Q1 Q2 Q3 Q4 Q5 Q6 Q7].^2;

% Estimate the path using the different model noises
for i = 1 : size(Q, 2) / size(X,1)
    Y = genNonLinearMeasurementSequence(X, h, R);
    [xf, Pf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q(:,
 5*(i-1) + 1 : 5*i), h, R, 'CKF');
    figure
    plot(X(1,:), X(2,:));
    hold on
    plot(xf(1,:), xf(2,:))
    legend('x', 'x_f')
    %[meas_x,meas_y] = convert_measurement_to_position(Y(1, :),
 Y(2, :), s1, s2);
    %plot(meas_x, meas_y);
    title(['Q = diag([',num2str(diag(Q(:, 5*(i-1) + 1 :
 5*i)).'), '])'])
end
```

3b

```
clc; clear; close all
% Set up parameters
rng(100)
[X, T] = truce_track();

x0 = zeros(5,1);
P0 = diag([100, 100, 100 (5*pi/180)^2 (1*pi/180)^2]);
s1 = [280, -80].';
s2 = [280, -200].';
R = (eye(2) * 4*pi/180).^2;

f = @(x)coordinatedTurnMotion(x, T);
h = @(x)dualBearingMeasurement(x, s1, s2);
Y = genNonLinearMeasurementSequence(X, h, R);

% Estimate and plot a good filter
Q = diag([0 0 0 0 pi/180*0.001]);
[xf, Pf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q, h,
 R, 'CKF');
plot(X(1,:), X(2,:))
hold on
plot(xf(1,:), xf(2,:))
legend('x', 'xf')
title('Good filter')
```

```
%plot(convert_measurement_to_position(Y(1,:), Y(2,:), s1, s2), 'o')


% 3c
% Estimate and plot a good filter
Q = diag([0 0 0 0 pi/180*0.001]);
[xf, Pf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q, h,
 R, 'CKF');
figure
plot(X(1,:), X(2,:))
hold on
plot(xf(1,:), xf(2,:))
for i = 1:10:size(xf,2)
   [ xy ] = sigmaEllipse2D( xf(1:2,i), Pf(1:2, 1:2, i), 3, 256 );
   plot(xy(1,:), xy(2,:), '--b')
end
legend('x', 'xf', '3\sigma')
title('Well tuned')


% Estimate and plot a under tuned filter
figure
Q = diag([0 0 0.000005 0 pi/180*0.000001]).^2;
[xf, Pf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q, h,
 R, 'CKF');
plot(X(1,:), X(2,:))
hold on
plot(xf(1,:), xf(2,:))
for i = 1:10:size(xf,2)
   [ xy ] = sigmaEllipse2D( xf(1:2,i), Pf(1:2, 1:2, i), 3, 256 );
   plot(xy(1,:), xy(2,:), '--b')
end
legend('x', 'xf', '3\sigma')
title('Under tuned')

% Estimate and plot a under over filter
figure
Q = diag([0 0 100 0 pi/1.80]).^2;
[xf, Pf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q, h,
 R, 'CKF');
plot(X(1,:), X(2,:))
hold on
plot(xf(1,:), xf(2,:))
for i = 1:10:size(xf,2)
   [ xy ] = sigmaEllipse2D( xf(1:2,i), Pf(1:2, 1:2, i), 3, 256 );
   plot(xy(1,:), xy(2,:), '--b')
end
legend('x', 'xf', '3\sigma')
title('Over tuned')

% Calculate the norm of the estimation error
figure
Q = diag([0 0 0 0 pi/180*0.001]);
```

```
[xf, Pf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q, h,
 R, 'CKF');
vals = [];
for i = 1 : size(xf,2)
   vals = [vals norm(X(1:2, i+1) - xf(1:2, i))];
end
plot(vals)
hold on
title('position error')
legend('|x-x_f|_2')




% 3d

function a = plot_every_fifth(x, P, level)
    for i = 1:5:size(x,2)
        [ xy ] = sigmaEllipse2D( x(1:2,i), P(1:2, 1:2, i), level,
 256 );
        plot(xy(1,:), xy(2,:), '--b')
    end
end

function [X, T] = 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 [X, Y, xf_ekf, Pf_ekf, xf_ukf, Pf_ukf, xf_ckf, Pf_ckf] =
 generate_data_and_estimate(x0, P0, f, Q, R, n, h)
X = genNonLinearStateSequence(x0, P0, f, Q, n);
Y = genNonLinearMeasurementSequence(X, h, R);
```

```
[xf_ekf, Pf_ekf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q, h,
 R, 'EKF');
[xf_ukf, Pf_ukf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q, h,
 R, 'UKF');
[xf_ckf, Pf_ckf, xp, Pp] = nonLinearKalmanFilter(Y, x0, P0, f, Q, h,
 R, 'CKF');

X = X(:, 2:end);
end

function [x,y] = convert_measurement_to_position(angle1, angle2, s1,
 s2)
t1 = tan(angle1);
t2 = tan(angle2);

s1y = s1(2);
s1x = s1(1);
s2y = s2(2);
s2x = s2(1);

y = (t2.*(-s1y+s1x.*t1-s2x.*t1) + t1.*s2y)./(t1-t2);
x = (y-s1(2))./t1 + s1(1);
end

function [x, y] = getPosFromMeasurement(y1, y2, s1, s2)
%GETPOSFROMMEASUREMENT computes the intersection point (transformed 2D
%measurement in Cartesian coordinate system) given two sensor
 locations and
%two bearing measurements, one from each sensor.
%INPUT: y1: bearing measurement from sensor 1 --- scalar
%       y2: bearing measurement from sensor 2 --- scalar
%       s1: location of sensor 1 in 2D Cartesian --- vector of length
 2
%       s2: location of sensor 2 in 2D Cartesian --- vector of length
 2
%OUTPUT: x: coordinate of intersection point on x axis
%        y: coordinate of intersection point on y axis

%This problem can be formulated as solving a set of two linear
 equations
%with two unknowns. Specifically, one would like to obtain (x,y) by
 solving
% (y-s1(2))=(x-s1(1))tan(y1) and (y-s2(2))=(x-s2(1))tan(y2).

x = (s2(2) - s1(2) + tan(y1)*s1(1) - tan(y2)*s2(1))/(tan(y1) -
 tan(y2));
y = s1(2) + tan(y1)*(x(1,:) - s1(1));

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 [xf, Pf, xp, Pp] = nonLinearKalmanFilter(Y, x_0, P_0, f, Q,
 h, R, type)
%NONLINEARKALMANFILTER 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
%                       [fx,Fx]=f(x)
%                       Takes as input x (state)
%                       Returns fx and Fx, motion model and Jacobian
 evaluated at x
%   Q           [n x n] Process noise covariance
%   h                   Measurement model function handle
```

```
%                       [hx,Hx]=h(x,T)
%                       Takes as input x (state),
%                       Returns hx and Hx, measurement model and
 Jacobian evaluated at x
%   R           [m x m] Measurement noise covariance
%
%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
%

% Your code here. If you have good code for the Kalman filter, you
 should re-use it here as
% much as possible.

% My code
%%%%%%%%%%%%%%%%%
% Parameters
N = size(Y,2);

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

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

% Predict one step ahead in time.
[x, p] = nonLinKFprediction(x_0, P_0, f, Q, type);
xp = x;
Pp(:,:,1) = p;

% Update the estimated position with the measurement
[x, p] = nonLinKFupdate(x, p, Y(:, 1), h, R, type);
xf = x;
Pf(:,:,1) = p;

for i = 2:size(Y,2)
    % Predict one step ahead in time.
    [x, p] = nonLinKFprediction(xf(:, end), Pf(:,:, i-1), f, Q, type);
    xp = [xp x];
    Pp(:,:,i) = p;

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


end
```

```
function [x, P, S, y_pred] = nonLinKFupdate(x, P, y, h, R, 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
%   h           Measurement model function handle
%               [hx,Hx]=h(x)
%               Takes as input x (state),
%               Returns hx and Hx, measurement model and Jacobian
 evaluated at x
%               Function must include all model parameters for the
 particular model,
%               such as sensor position for some models.
%   R           [m x m] Measurement noise covariance
%   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'

            % Your EKF update here
            %%%%%%%%%%%%%%%%%%%%%%
            [hx,Hx]=h(x);
            y_pred = hx;
            S = Hx*P*Hx.' + R; % Predict the covariance in yk
            K = P*Hx.'*S^-1;  % Calculate the Kalman gain, how much we
 trust the new measurement
            P = P - K*S*K.'; %Estimate the error covariance
            x = x + K*(y-hx); % Estimate the new state
            %%%%%%%%%%%%%%%%%%%%%%

        case 'UKF'

            % Your UKF update here
            %%%%%%%%%%%%%%%%%%%%%
            [SP,W] = sigmaPoints(x, P, type);
            % Predict y
            y_pred = 0;
            for i = 1 : size(W,2)
                [hx,Hx] = h(SP(:, i));
                y_pred = y_pred + hx*W(i);
            end
            % Estimate x covariance
            Pxy = 0;
            S = 0;
            for i = 1 : size(W,2)
```

```
                [hx, Hx] = h(SP(:, i));
                Pxy = Pxy + ((SP(:,i) - x)*(hx - y_pred).')*W(i);
                S = S + (hx - y_pred)*(hx - y_pred).'*W(:,i);
            end
            S = S + R;
            P = P - Pxy*S^-1*Pxy.';

            % Estimate x
            x = x + Pxy*S^-1*(y-y_pred);
            %%%%%%%%%%%%%%%%%%%%%

            % Make sure the covariance matrix is semi-definite
            if min(eig(P))<=0
                [v,e] = eig(P, 'vector');
                e(e<0) = 1e-4;
                P = v*diag(e)/v;
            end

        case 'CKF'

            % Your CKF update here
            %%%%%%%%%%%%%%%%%%%%%
            [SP,W] = sigmaPoints(x, P, type);
            % Predict y
            y_pred = 0;
            for i = 1 : size(W,2)
                [hx,Hx] = h(SP(:, i));
                y_pred = y_pred + hx*W(i);
            end

            % Estimate x covariance
            Pxy = 0;
            S = 0;
            for i = 1 : size(W,2)
                [hx, Hx] = h(SP(:, i));
                Pxy = Pxy + ((SP(:,i) - x)*(hx - y_pred).')*W(i);
                S = S + (hx - y_pred)*(hx - y_pred).'*W(:,i);
            end
            S = S + R;
            P = P - Pxy*S^-1*Pxy.';

            % Estimate x
            x = x + Pxy*S^-1*(y-y_pred);
            %%%%%%%%%%%%%%%%%%%%%
        otherwise
            error('Incorrect type of non-linear Kalman filter')
    end

end

function [x, P] = nonLinKFprediction(x, P, f, Q, 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
%               [fx,Fx]=f(x)
%               Takes as input x (state),
%               Returns fx and Fx, motion model and Jacobian evaluated
 at x
%               All other model parameters, such as sample time T,
%               must be included in the function
%   Q           [n x n] Process noise covariance
%   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'

            % Your EKF code here
            %%%%%%%%%%%%%%%%%%%%
            % Prediction step
            [fx,Fx]=f(x);
            % x mean
            x = fx;
            % x covariance
            P = Fx*P*Fx.' + Q;
            %%%%%%%%%%%%%%%%%%%%
        case 'UKF'

            % Your UKF code here
            %%%%%%%%%%%%%%%%%%%%
            [SP,W] = sigmaPoints(x, P, type);

            % Predict x mean
            x = 0;
            P = 0;
            for i = 1 : size(W,2)
                [fx,Fx]=f(SP(:, i));
                x = x + fx*W(i);
            end
            % Predict x covariance
            for i = 1 : size(W,2)
                [fx,Fx]=f(SP(:, i));
                P = P + ((fx-x)*(fx-x).')*W(i);
            end
            P = P + Q;
            %%%%%%%%%%%%%%%%%%%%

            % Make sure the covariance matrix is semi-definite
            if min(eig(P))<=0
                [v,e] = eig(P, 'vector');
```

```
                e(e<0) = 1e-4;
                P = v*diag(e)/v;
            end

        case 'CKF'

            % Your CKF code here
            %%%%%%%%%%%%%%%%%%%%
            [SP,W] = sigmaPoints(x, P, type);

            % Predict x mean
            x = 0;
            P = 0;
            for i = 1 : size(W,2)
                [fx,Fx]=f(SP(:, i));
                x = x + fx*W(i);
            end
            % Predict x covariance
            for i = 1 : size(W,2)
                [fx,Fx]=f(SP(:, i));
                P = P + ((fx-x)*(fx-x).')*W(i);
            end
            P = P + Q;
            %%%%%%%%%%%%%%%%%%%%
        otherwise
            error('Incorrect type of non-linear Kalman filter')
    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] UKF, [n x 2n] CKF. Matrix with sigma points
%   W           [1 x 2n+1] UKF, [1 x 2n] UKF. Vector with sigma point
 weights
%
    switch type
        case 'UKF'

            % your code
            %%%%%%%%%%%%%%%
            n = size(x, 1);

            % Calculate the SP weights
            w0 = 1-n/3;
            W = [w0 (1-w0)/(2*n)*ones(1, 2*n)];
```

```matlab
            % Generate the SP locations/values
            P_sqrt = sqrtm(P);
            SP = [x...
                x+sqrt(n/(1-w0))*P_sqrt...
                x-sqrt(n/(1-w0))*P_sqrt];
            %%%%%%%%%%%%%%%

        case 'CKF'

            % your code
            %%%%%%%%%%%%%%%
            n = size(x, 1);

            % Generate the SP locations/values
            P_sqrt = sqrtm(P);
            SP = [x+sqrt(n)*P_sqrt, x-sqrt(n)*P_sqrt];

            % Calculate the SP weights
            W = 1/(2*n)*ones(1, 2*n);
            %%%%%%%%%%%%%%%

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

end

function Y = genNonLinearMeasurementSequence(X, 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
%   h           Measurement model function handle
%               [hx,Hx]=h(x)
%               Takes as input x (state)
%               Returns hx and Hx, measurement model and Jacobian
 evaluated at x
%   R           [m x m] Measurement noise covariance
%
%Output:
%   Y           [m x N] Measurement sequence
%

% Your code here
%%%%%%%%%%%%%%%%%%
% Remove the first state as we don't want to measure that
X = X(:, 2:end);
Y = [];
for i = 1:size(X,2)
   % Measure the next state
   y = h(X(:, i));
   % Add noise to the measurement
```

```
    y = y + mvnrnd(zeros(size(y, 1), 1), R).';
    % Save the measurement
    Y = [Y y];
end
%%%%%%%%%%%%%%%%%%

end

function X = genNonLinearStateSequence(x_0, P_0, f, Q, N)
%GENLINEARSTATESEQUENCE generates an N+1-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
%               [fx,Fx]=f(x)
%               Takes as input x (state),
%               Returns fx and Fx, motion model and Jacobian evaluated
 at x
%               All other model parameters, such as sample time T,
%               must be included in the function
%   Q           [n x n] Process noise covariance
%   N           [1 x 1] Number of states to generate
%
%Output:
%   X           [n x N+1] State vector sequence
%

% Your code here
% Generate first state
X = mvnrnd(x_0, P_0).';
for i=1:N
    % Generate next state
    [fx,Fx]=f(X(:, end));
    % Append the new state to the state vector and apply motion noise
 to it
    X = [X fx+mvnrnd(zeros(size(Q,2), 1), Q).'];
end

end

function [hx, Hx] = dualBearingMeasurement(x, s1, s2)
%DUOBEARINGMEASUREMENT calculates the bearings from two sensors,
 located in
%s1 and s2, to the position given by the state vector x. Also returns
 the
%Jacobian of the model at x.
%
%Input:
%   x           [n x 1] State vector, the two first element are 2D
 position
%   s1          [2 x 1] Sensor position (2D) for sensor 1
```

```
%   s2          [2 x 1] Sensor position (2D) for sensor 2
%
%Output:
%   hx          [2 x 1] measurement vector
%   Hx          [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.

% Your code here
% Calculate y+, without the noise
hx = [atan2(x(2,:)-s1(2), x(1,:)-s1(1))
      atan2(x(2,:)-s2(2), x(1,:)-s2(1))];

% Calculate deriv(y+) evaluated at x, without the noise
Hx = [-(x(2,:)-s1(2,:))./((x(2,:)-s1(2,:)).^2 + (x(1,:)-s1(1,:)).^2)
 (x(1,:)-s1(1,:))./((x(2,:)-s1(2,:)).^2 + (x(1,:)-s1(1,:)).^2)
      -(x(2,:)-s2(2,:))./((x(2,:)-s2(2,:)).^2 + (x(1,:)-s2(1,:)).^2)
 (x(1,:)-s2(1,:))./((x(2,:)-s2(2,:)).^2 + (x(1,:)-s2(1,:)).^2)];
% Append zeros to the end cuz the rest of the states don't depent on
 x(1) or x(2)
Hx = [Hx zeros(2, size(x,1)-size(Hx,1))];
end

function [fx, Fx] = 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:
%   fx          [5 x 1] motion model evaluated at state x
%   Fx          [5 x 5] motion model Jacobian evaluated at state x
%
% 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

% Your code for the motion model here
% Calculate the next state vector x, dissregarding the noise:
% x+ = [x+, y+, v+, theta+, omega+]
fx = [x(1) + T*x(3)*cos(x(4))
      x(2) + T*x(3)*sin(x(4))
      x(3)
      x(4) + T*x(5)
```

```
      x(5)];

%Check if the Jacobian is requested by the calling function
if nargout > 1
    % Your code for the motion model Jacobian here
    % F(x) is the derivation of x+ with respect to x evaluated at xk
    Fx = [1 0 T*cos(x(4)) -T*x(3)*sin(x(4)) 0
          0 1 T*sin(x(4)) T*x(3)*cos(x(4)) 0
          0 0 1 0 0
          0 0 0 1 T
          0 0 0 0 1];
end

end
```

Published with MATLAB® R2019b