```
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] = 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);
            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)];

            % 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