SSY345 Sensor fusion and non linear filtering

HA2 Implementation

Isak Åslund (isakas)

May 5, 2020

Matlab code

coordinatedTurnMotion.m

```
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
%   theta       heading
%   omega       turn-rate

% For easy visualisation, pull out the members and assign appropriate names
px = x(1);
py = x(2);
v = x(3);
theta = x(4);
omega = x(5);


% Next time step x(k) = f(x(k-1)) + q(k-1)
fx = [
  px + T*v*cos(theta);
  py + T*v*sin(theta);
  v;
  theta + T*omega;
  omega;
];

%Check if the Jacobian is requested by the calling function
if nargout > 1
```

```
    Fx = [
        1    0    T*cos(theta)     -T*v*sin(theta) 0;
        0    1    T*sin(theta)     T*v*cos(theta)  0;
        0    0    1                0               0;
        0    0    0                1               T;
        0    0    0                0               1
    ];
end

end
```

dualBearingMeasurement.m

```
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.
px = x(1);
py = x(2);

n = length(x);

% Your code here
hx = [
    atan2(py-s1(2), px-s1(1));
    atan2(py-s2(2), px-s2(1));
];


Hx = [
    -(py-s1(2)) /  ( (px-s1(1))^2 + (py-s1(2))^2),   (px-s1(1)) / ((py-s1(2))^2 + (px
    -(py-s2(2)) /  ( (px-s2(1))^2 + (py-s2(2))^2),   (px-s2(1)) / ((py-s2(2))^2 + (px
];



end
```

genNonLinearMeasurementSequence.m

```
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
%

% Pre allocate for measurement
Y = zeros(size(R,1), size(X,2)-1);

% Create measurement data from all states except the first
for k = 1:size(Y,2)
   Y(:,k) = mvnrnd(h(X(:,k+1)), R, 1);
end

end
```

genNonLinearStateSequence.m

```
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
%

%Generate initial state from prior
x0 = mvnrnd(x_0, P_0, 1)';

% Generate state sequence by letting the initial state propagate
X = zeros(length(x_0), N+1);  % State sequence pre allocation
X(:,1) = x0;                  % Add initial state

for k = 2:N+1
    X(:,k) = mvnrnd( f(X(:,k-1)), Q, 1);
end


end
```

nonLinearKalmanFilter.m

```
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 a
%   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
%

% Parameters
N = size(Y,2);

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

% Data allocation
%X = zeros(n,N);
%P = zeros(n,n,N);
%V = zeros(1,N);

% 1. Predict the next state
% 2. Update the prediction with measurement
for k = 1:N
    % Prediction
```

```
    [x_0, P_0] = nonLinKFprediction(x_0, P_0, f, Q, type);
    xp(:,k) = x_0;
    Pp(:,:,k) = P_0;

    % Update
    [x_0, P_0] = nonLinKFupdate(x_0, P_0, Y(:,k), h, R, type);
    xf(:,k) = x_0;
    Pf(:,:,k) = P_0;



end


end
```

nonLinKFprediction.m

```
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
%

% Prediction
%[fx, Fx] = f(x);
n = size(x,1);
switch type
    case 'EKF'

        % 1. Predict state using non linear function
        % 2. Estimate the gaussian covariance using a linearization
        % Note: The acctual distrubution is not gaussian but we approximate
        %       to a gaussian to simplify.
        [fx, Fx] = f(x);
        x = fx;
        P = Fx*P*Fx'+ Q;



    case 'UKF'
        % 1. Get sigma points
        % 2. Transform SP using non linear transform
        % 3. Calculate estimated mean using transformed SP
```

```
    % 4. Calculate estimated covar using transformed SP and mean

    [SP,W] = sigmaPoints(x, P, 'UKF');
    %Transform sigma points
    for i = 1:length(SP)
        fx(:,i) = f(SP(:,i));
    end

    x = 0;
    for i = 1:(2*n+1)
        x = x + W(i).*fx(:,i);
    end

    P = zeros(n,n);
    for i = 1:(2*n+1)
        P = P + ( (fx(:,i)-x) * (fx(:,i)-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'

    % 1. Get sigma points
    % 2. Transform SP using non linear transform
    % 3. Calculate estimated mean using transformed SP
    % 4. Calculate estimated covar using transformed SP and mean

    [SP,W] = sigmaPoints(x, P, 'CKF');
    %Transform sigma points
    for i = 1:length(SP)
        fx(:,i) = f(SP(:,i));
    end

    x = 0;
    for i = 1:(2*n)
        x = x + W(i).*fx(:,i);
    end
```

```
        P = zeros(n,n);
        for i = 1:(2*n)
            P = P + W(i).*((fx(:,i)-x) * (fx(:,i)-x)');
        end
        P = P + Q;

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

end
```

nonLinKFupdate.m

```
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
%
m = length(y);
n = length(x);
switch type
    case 'EKF'
        [hx Hx] = h(x);
        S = Hx*P*Hx' + R;
        K = P*Hx'*inv(S);
        x = x + K*(y - hx);
        P = P - K*S*K';

    case 'UKF'
        [SP, W] = sigmaPoints(x,P,type);
        for i = 1:length(SP)
            hx(:,i) = h(SP(:,i));
        end

        % Pre allocation
        y_pred = 0;
        P_xy = 0;
        S = R;
```

```
    for i = 1:(2*n+1)
        y_pred = y_pred + W(i).*hx(:,i);
    end
    for i = 1:(2*n+1)
        S = S +  (hx(:,i)-y_pred)*(hx(:,i)-y_pred)' .* W(i);
        P_xy = P_xy + ( (SP(:,i)-x) * (hx(:,i)-y_pred)' ) .*W(i);
    end

    %Measurement update
    x = x + P_xy*inv(S)*(y - y_pred);
    P = P - P_xy*inv(S)*P_xy';

     % 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'
    [SP, W] = sigmaPoints(x,P,type);
    for i = 1:length(SP)
        hx(:,i) = h(SP(:,i));
    end

    % Pre allocation
    y_pred = 0;
    P_xy = 0;
    S = R;
    for i = 1:(2*n)
        y_pred = y_pred + W(i).*hx(:,i);
    end
    for i = 1:(2*n)
        S = S +  (hx(:,i)-y_pred)*(hx(:,i)-y_pred)' .* W(i);
        P_xy = P_xy + ( (SP(:,i)-x) * (hx(:,i)-y_pred)' ) .*W(i);
    end

    %Measurement update
    x = x + P_xy*inv(S)*(y - y_pred);
    P = P - P_xy*inv(S)*P_xy';

    if min(eig(P))<=0
          [v,e] = eig(P, 'vector');
         e(e<0) = 1e-4;
```

```
            P = v*diag(e)/v;
        end

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

end
```

samplesToState.m

```
function [x, y] = samplesToState(measurements, s1, s2)
    t1 = measurements(1,:);
    t2 = measurements(2,:);

    x = (tan(t1)*s1(1) - tan(t2)*s2(1) + s2(2) - s1(2)) ./ (tan(t1) - tan(t2));
    y = tan(t1).*(x - s1(1)) + s1(2);
end
```

sigmaEllipse2D.m

```
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

    %Your code here

    %Evenly spaced points
    theta = linspace(0, 2*pi, npoints);

    %Level curve
    xy = mu + level*sqrtm(Sigma) * [cos(theta); sin(theta)];

end
```

sigmaPoints.m

```
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
%
n = size(x,1);
P_sqrt = sqrtm(P);

switch type
    case 'UKF'
        num_points = 2*n + 1;

        SP = zeros(n, num_points);
        SP(:,1) = x;

        W = zeros(1, num_points);
        W0 = 1 - n/3;

        W(1) = W0;
        SP(:,1) = x;

        for i = 1:n
           W(i+1) = (1-W0) / (2*n);
           W(i+1+n) = (1-W0) / (2*n);

           SP(:,i+1)   = x + sqrt(n/(1-W0))*P_sqrt(:,i);
           SP(:,i+1+n) = x - sqrt(n/(1-W0))*P_sqrt(:,i);
        end


    case 'CKF'
        num_points = 2*n;

        SP = zeros(n, num_points);
```

```
        W = zeros(1, num_points);

        for i = 1:n
            W(i)       = 1/(2*n);
            W(i+n)     = W(i);
            SP(:,i)    = x + sqrt(n)*P_sqrt(:,i);
            SP(:,i+n)  = x - sqrt(n)*P_sqrt(:,i);
        end

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

end
```