SSY191 - Sensor Fusion and Nonlinear Filtering
Implementation of Home Assignment 03

Lucas Rath

Listings


```
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

    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

        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
```

```
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.


    % Procedure to calculate hx and Hx using symbolic toolbox

% % %     % states
% % %     syms px py real
% % %     x = [px py].'
% % %
% % %     % sensor positions
% % %     s1 = sym('s1_',[2;1],'real')
% % %     s2 = sym('s2_',[2;1],'real')
% % %
% % %     % sensor readings
% % %     ang1 = atan2( py-s1(2), px-s1(1) )
% % %     ang2 = atan2( py-s2(2), px-s2(1) )
% % %
% % %     hx = [ang1; ang2]
% % %     Hx = jacobian(hx,x)

    % initialize outputs with correct sizes
    n  = size(x,1);
    hx = zeros(2,1);
    Hx = zeros(2,n);

    % calculate readings from the two sensors
    ang1 = atan2( x(2)-s1(2), x(1)-s1(1) );
    ang2 = atan2( x(2)-s2(2), x(1)-s2(1) );

    % output is the concatenation of the two readings
    hx(1:2,1) = [ang1;
                 ang2];

    % jacobian of hx, as calculated using the symbolic toolbox
    Hx(1:2,1:2) = [-(x(2)-s1(2)) / ( (x(1)-s1(1))^2 + (x(2)-s1(2))^2 ),...
                     (x(1)-s1(1)) / ((x(1)-s1(1))^2 + (x(2)-s1(2))^2);
                   -(x(2)-s2(2)) / ( (x(1)-s2(1))^2 + (x(2)-s2(2))^2 ), ...
                     (x(1)-s2(1)) / ((x(1)-s2(1))^2 + (x(2)-s2(2))^2)];
end
```

```
function X = genNonLinearStateSequence(x_0, P_0, f, Q, N)
    % GENNONLINEARSTATESEQUENCE 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

    n = length(x_0);
    X = zeros(n,N);

    % sample initial state from the prior distribution x0~N(x_0,P_0)
    X(:,1) = mvnrnd(x_0, P_0)';
    % iterate to generate N samples
    for i=1:N
        % Motion model: X{k} = f(X{k-1}) + q{k-1},   where q{k-1} ~ N(0,Q)
        X(:,i+1) = f(X(:,i)) + mvnrnd(zeros(n,1), Q)';
    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
    %   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

    m = size(R,1);
    % state sequence includes x0, which does not generate an observation
    N = size(X,2) -1;
    Y = zeros(m,N);

    % iterate to generate N samples
    for i=1:N
        % Measurement model: Y{k} = h(X{k}) + r{k},   where r{k} ~ N(0,R)
        Y(:,i) = h(X(:,i+1)) + mvnrnd(zeros(m,1), 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] 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);
    switch type
        case 'UKF'
            % initialize outputs
            SP = zeros(n, 2*n+1);
            % calculate covariance square root
            Psqrt = sqrtm(P);
            W0 = 1 - n/3;
            % calculate sigma points
            SP(:,1) = x;
            for i=1:n
                SP(:, i+1  ) = x + sqrt( n / (1 - W0) ) * Psqrt(:,i);
                SP(:, i+n+1) = x - sqrt( n / (1 - W0) ) * Psqrt(:,i);
            end
            % calculate weights
            W  = [W0, (1-W0)/(2*n)*ones(1, 2*n)];
        case 'CKF'
            % initialize outputs
            SP = zeros(n, 2*n);
            % calculate covariance square root
            Psqrt = sqrtm(P);
            % calculate sigma points
            for i=1:n
                SP(:, i  ) = x + sqrt(n) * Psqrt(:,i);
                SP(:, i+n) = x - sqrt(n) * Psqrt(:,i);
            end
            % calculate weights
            W  = 1/(2*n) * ones(1, 2*n);
        otherwise
            error('Incorrect type of sigma point')
    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

    n = size(x,1);

    if strcmp(type,'EKF')
        % calculate f(x) and jacobian(f(x),x)
        [fx, dfx] = f(x);
        % predict using first order Taylor expansion
        x = fx;
        P = dfx * P * dfx' + Q;
    elseif strcmp(type,'UKF') || strcmp(type,'CKF')
        % compute sigma points
        [SP,W] = sigmaPoints(x, P, type);
        % predict mean
        x = zeros(n,1);
        for i=1:numel(W)
            x = x + f(SP(:,i)) * W(i);
        end
        % predict covariance
        P = Q;
        for i=1:numel(W)
            P = P + (f(SP(:,i))-x)*(f(SP(:,i))-x).' * W(i);
        end
        % 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
    else
        error('Incorrect type of non-linear Kalman filter')
    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] Predicted mean
    %   P           [n x n] Predicted 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

    n = size(x,1);

    if strcmp(type,'EKF')
        % calculate h(x) and jacobian(h(x),x)
        [hx, dhx] = h(x);
        % calculate inovation covariance and kalman gain
        S = dhx * P * dhx' + R;
        K = P * dhx' / S;
        % update
        x = x + K * ( y - hx );
        P = P - K * S * K';
    elseif strcmp(type,'UKF') || strcmp(type,'CKF')
        % compute sigma points
        [SP,W] = sigmaPoints(x, P, type);

        % estimate desired moments
        yhat = 0*y;
        for i=1:numel(W)
            yhat = yhat + h(SP(:,i)) * W(i);
        end
        Pxy = 0*x*y';
        for i=1:numel(W)
            Pxy = Pxy + (SP(:,i)-x)*(h(SP(:,i))-yhat)' * W(i);
        end
        S = R;
        for i=1:numel(W)
            S = S + (h(SP(:,i))-yhat)*(h(SP(:,i))-yhat)' * W(i);
        end

        % calculate updated mean and covariance
        x = x + Pxy / S * ( y - yhat );
        P = P - Pxy / S * Pxy';

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

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

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

    %% Data allocation

    xp = zeros(n,   N);
    Pp = zeros(n,n, N);

    xf = zeros(n,   N +1);
    Pf = zeros(n,n, N +1);

    %% filter
    xf(:,1)   = x_0;
    Pf(:,:,1) = P_0;

    for i=1:N
        % prediction step: compute p(x_k | y_1:k-1) from p(x_k-1 | y_1:k-1)
        [xp(:,i), Pp(:,:,i)] = nonLinKFprediction(xf(:,i), Pf(:,:,i), f, Q, type);
        % update step: compute p(x_k | y_1:k) from p(x_k | y_1:k-1)
        [xf(:,i+1), Pf(:,:,i+1)] = nonLinKFupdate(xp(:,i), Pp(:,:,i), Y(:,i), h, R, type);
    end

    % exclude prior x0~N(x_0, P_0) from posterior
    xf = xf(:,2:end);
    Pf = Pf(:,:,2:end);

end
```