SSY191 - Sensor Fusion and Nonlinear Filtering
Implementation of Home Assignment 04

Lucas Rath

Listings

```
function [xs, Ps, xf, Pf, xp, Pp] = nonLinRTSsmoother(Y, x_0, P_0, f, Q, 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


    %% Parameters
    N = size(Y,2);
    n = length(x_0);
    m = size(Y,1);


    %% Forward filtering

%     f2 = @(x) f(x,T);
%     h2 = @(x) h(x,S(:,1));
    [xf, Pf, xp, Pp] = nonLinearKalmanFilter(Y, x_0, P_0, f, Q, h, R, type);

    %% Backward filtering

    % initialize outputs
    xs(:,N) = xf(:,N);
    Ps(:,:,N) = Pf(:,:,N);

    % backward recursion - last filter output is already smoothed
    for k=N-1:-1:1
        [xs(:,k), Ps(:,:,k)] = nonLinRTSSupdate(xs(:,k+1), Ps(:,:,k+1), xf(:,k), Pf(:,:,k), xp(:,k+1
Pp(:,:,k+1), f, sigmaPoints, type);
    end

end
```

```matlab
function [xs, Ps] = nonLinRTSSupdate(xs_kplus1, ...
                                     Ps_kplus1, ...
                                     xf_k, ...
                                     Pf_k, ...
                                     xp_kplus1, ...
                                     Pp_kplus1, ...
                                     f, ...
                                     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


    %% calculate Pkkp1 = P_{k,k+1|k}

    if strcmp( type, 'EKF' )
        % evaluate motion model at ^x_{k|k}
        [~, df_xk] = f(xf_k);
        % approximate P_{k,k+1|k}
        Pkkp1 = Pf_k * df_xk';

    else % UKF or CKF
        % calcualte sigma points of p(x_{k|k})
        [SP,W] = sigmaPoints(xf_k, Pf_k, type);
        % predict covariance Pkkp1 = P_{k,k+1|k}
        Pkkp1 = zeros(size(xf_k,1));
        for i=1:numel(W)
            Pkkp1 = Pkkp1 + (SP(:,i)-xf_k)*(f(SP(:,i))-xp_kplus1).' * W(i);
        end
    end

    %% backward recursion

    % calculate smoothing gain
    Gk =  Pkkp1 / Pp_kplus1;
    % calculate Smoothed estimate and covariance at time k
    xs = xf_k + Gk * ( xs_kplus1 - xp_kplus1 );
    Ps = Pf_k - Gk * ( Pp_kplus1 - Ps_kplus1 ) * Gk';

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

    N=size(X,2);

    % Generates the segmented numberline from 0 to 1 (upper edge not included since it will never be
    segment = [0 cumsum(W)];

    % draw samples from uniform distribution on [0,1]
    samples = rand([1 N]);

    j=zeros(1,N);
    for i=1:N
        j(i) = find(samples(i) >= segment,1,'last');
    end

    Wr = 1/N*ones(1,N);
    Xr = X(:,j);
end
```

```
function [X_k, W_k] = pfFilterStep(X_kmin1, W_kmin1, y_k, proc_f, proc_Q, meas_h, meas_R)
    % 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

    % calculate p(x(i)_k|x(i)_{k-1})
    X_k = mvnrnd( proc_f(X_kmin1)' ,proc_Q )';

    % calculate p(y_k|x(i)_k)
    Wy = mvnpdf(y_k',meas_h(X_k)',meas_R)';

    % compute weights
    W_k = W_kmin1 .* Wy;
    W_k = W_k / sum(W_k);
end
```

```
function [xfp, Pfp, Xp, Wp] = pfFilter(x_0, P_0, Y, proc_f, proc_Q, meas_h, meas_R, N, bResample, pl
    % PFFILTER Filters measurements Y using the SIS or SIR algorithms and a
    % state-space model.
    %
    % Input:pfFilter(x_0
    %   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


    n = size(x_0,1);
    K = size(Y,2);

    % allocate memory
    xfp = zeros(n,K);
    Pfp = zeros(n,n,K);
    Xp = zeros(n,N,K);
    Wp = zeros(N,K);

    % sample initial particles around prior distribution
    Xp(:,:,1) = mvnrnd(x_0,P_0,N)';
    Wp(:,1)   = 1/N * ones(1,N);

    j = 1:N;
    for k=2:K+1
        % perform a particle filter step for the next measurement
        [Xp(:,:,k), Wp(:,k)] = pfFilterStep(Xp(:,:,k-1), Wp(:,k-1)', Y(:,k-1), proc_f, proc_Q, meas_
        plotFunc(k-1, Xp(:,:,k), Xp(:,:,k-1), Wp(:,k)', j);
        % resample
        if bResample
            [Xp(:,:,k), Wp(:,k), j] = resampl(Xp(:,:,k), Wp(:,k)');
        end
        % estimate mean and covariance given the particles
        xfp(:,k)   = sum( Xp(:,:,k).*Wp(:,k)' ,2 );
        Pfp(:,:,k) = Wp(:,k)'.*(Xp(:,:,k) - xfp(:,k))*(Xp(:,:,k) - xfp(:,k))';
    end

    % remove prior from vector
    xfp = xfp(:,2:end);
    Pfp = Pfp(:,:,2:end);
    Xp  = Xp(:,:,2:end);
    Wp  = Wp(:,2:end);
end
```