HA4- Implementation

Arshad Nowsath

1.Non-Linear RTS smoother update

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


% Your code here.
switch type
    case 'EKF'
    [fx, Fx]=f(xf_k,T);
    P_kkpl=Pf_k*Fx';
    innov = xs_kplus1-xp_kplus1;

    case 'UKF'
        [SP,W]=sigmaPoints(xf_k, Pf_k,type);
```

```
        P_kkpl=0;
        for i=1:length(SP)
            P_kkpl=P_kkpl+(SP(:,i)-xf_k)*(f(SP(:,i),T)-xp_kplus1)'*W(i);
        end
        innov=xs_kplus1-xp_kplus1;
    case 'CKF'
        [SP,W]=sigmaPoints(xf_k, Pf_k,type);
        P_kkpl=0;
        for i=1:length(SP)
            P_kkpl=P_kkpl+(SP(:,i)-xf_k)*(f(SP(:,i),T)-xp_kplus1)'*W(i);
        end
        innov=xs_kplus1-xp_kplus1;
    otherwise
        error('Something went wrong with the types')
end
G_k = P_kkpl * inv(Pp_kplus1);
xs = xf_k + G_k * innov;
Ps = Pf_k - G_k * (Pp_kplus1 - Ps_kplus1) * G_k';
end
```

2.Non-Linear RTS smoother

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


% your code here!
% Parameters
N = size(Y,2);
n = length(x_0);




% Data allocation
xf = zeros(n,N+1);
Pf = zeros(n,n,N+1);
xp = zeros(n,N);
Pp = zeros(n,n,N);


% KalmanFilter
% Prior knowledge of state
xf(:,1) = x_0;
Pf(:,:,1) = P_0;


for i = 1:N
    % Prediction step:
    [xp(:,i), Pp(:,:,i)] = nonLinKFprediction(xf(:,i), Pf(:,:,i), f, T, Q,
sigmaPoints, type);
    % Update step:
    [xf(:,i+1), Pf(:,:,i+1)] = nonLinKFupdate(xp(:,i), Pp(:,:,i), Y(:,i),
S(:,i), h, R, sigmaPoints, type);
end
xf = xf(:,2:end);
Pf = Pf(:,:,2:end);


% Intial values
xs(:,N) = xf(:,N);
Ps(:,:,N) = Pf(:,:,N);


for i = N-1:-1:1
% Smoothing update step:
[xs(:,i), Ps(:,:,i)] = nonLinRTSSupdate(xs(:,i+1), Ps(:,:,i+1), xf(:,i),
Pf(:,:,i), xp(:,i+1), Pp(:,:,i+1), f, T, sigmaPoints, type);
```

```
end
% We have offered you functions that do the non-linear Kalman prediction and
update steps.
% Call the functions using
% [xPred, PPred] = nonLinKFprediction(x_0, P_0, f, T, Q, sigmaPoints, type);
% [xf, Pf] = nonLinKFupdate(xPred, PPred, Y, S, h, R, sigmaPoints, type);
end
function [xs, Ps] = nonLinRTSSupdate(xs_kplus1, ...
                                     Ps_kplus1, ...
                                     xf_k, ...
                                     Pf_k, ...
                                     xp_kplus1, ...
                                     Pp_kplus1, ...
                                     f, ...
                                     T, ...
                                     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


% Your code here.
switch type
    case 'EKF'
    [fx, Fx]=f(xf_k,T);
    P_kkpl=Pf_k*Fx';
    innov = xs_kplus1-xp_kplus1;

    case 'UKF'
```

```
        [SP,W]=sigmaPoints(xf_k, Pf_k,type);
        P_kkpl=0;
        for i=1:length(SP)
            P_kkpl=P_kkpl+(SP(:,i)-xf_k)*(f(SP(:,i),T)-xp_kplus1)'*W(i);
        end
        innov=xs_kplus1-xp_kplus1;
    case 'CKF'
        [SP,W]=sigmaPoints(xf_k, Pf_k,type);
        P_kkpl=0;
        for i=1:length(SP)
            P_kkpl=P_kkpl+(SP(:,i)-xf_k)*(f(SP(:,i),T)-xp_kplus1)'*W(i);
        end
        innov=xs_kplus1-xp_kplus1;
    otherwise
        error('Something went wrong with the types')
end
G_k = P_kkpl * inv(Pp_kplus1);
xs = xf_k + G_k * innov;
Ps = Pf_k - G_k * (Pp_kplus1 - Ps_kplus1) * G_k';
end
```

```
function [x, P] = nonLinKFprediction(x, P, f, T, Q, sigmaPoints, 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
%   T           Sampling time
%   Q           [n x n] Process noise covariance
%   sigmaPoints Handle to function that generates sigma points.
%   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'
```

```
            % Evaluate motion model
            [fx, Fx] = f(x,T);
            % State prediction
            x = fx;
            % Covariance prediciton
            P = Fx*P*Fx' + Q;
            % Make sure P is symmetric
            P = 0.5*(P + P');


        case 'UKF'


            % Predict
            [x, P] = predictMeanAndCovWithSigmaPoints(x, P, f, T, Q,
sigmaPoints, type);


            if min(eig(P))<=0
                [v,e] = eig(P);
                emin = 1e-3;
                e = diag(max(diag(e),emin));
                P = v*e*v';
            end


        case 'CKF'


            % Predict
            [x, P] = predictMeanAndCovWithSigmaPoints(x, P, f, T, Q,
sigmaPoints, type);


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


function [x, P] = nonLinKFupdate(x, P, y, s, h, R, sigmaPoints, 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
%   s           [2 x 1] sensor position vector
%   h           Measurement model function handle
%   R           [n x n] Measurement noise covariance
%   sigmaPoints Handle to function that generates sigma points.
%   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'

        % Evaluate measurement model
        [hx, Hx] = h(x,s);

        % Innovation covariance
        S = Hx*P*Hx' + R;
        % Kalman gain
        K = (P*Hx')/S;

        % State update
        x = x + K*(y - hx);
        % Covariance update
        P = P - K*S*K';

        % Make sure P is symmetric
        P = 0.5*(P + P');

    case 'UKF'


        % Update mean and covariance
        [x, P] = updateMeanAndCovWithSigmaPoints(x, P, y, s, h, R,
sigmaPoints, type);

        if min(eig(P))<=0
            [v,e] = eig(P);
            emin = 1e-3;
            e = diag(max(diag(e),emin));
```

```
            P = v*e*v';
        end

    case 'CKF'


        % Update mean and covariance
        [x, P] = updateMeanAndCovWithSigmaPoints(x, P, y, s, h, R,
sigmaPoints, type);

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


end




function [x, P] = predictMeanAndCovWithSigmaPoints(x, P, f, T, Q, sigmaPoints,
type)
%
%PREDICTMEANANDCOVWITHSIGMAPOINTS computes the predicted mean and covariance
%
%Input:
%   x           [n x 1] mean vector
%   P           [n x n] covariance matrix
%   f           measurement model function handle
%   T           sample time
%   Q           [m x m] process noise covariance matrix
%
%Output:
%   x           [n x 1] Updated mean
%   P           [n x n] Updated covariance
%


    % Compute sigma points
    [SP,W] = sigmaPoints(x, P, type);


    % Dimension of state and number of sigma points
    [n, N] = size(SP);
```

```
    % Allocate memory
    fSP = zeros(n,N);

    % Predict sigma points
    for i = 1:N
        [fSP(:,i),~] = f(SP(:,i),T);
    end

    % Compute the predicted mean
    x = sum(fSP.*repmat(W,[n, 1]),2);

    % Compute predicted covariance
    P = Q;
    for i = 1:N
        P = P + W(i)*(fSP(:,i)-x)*(fSP(:,i)-x)';
    end

    % Make sure P is symmetric
    P = 0.5*(P + P');

end

function [x, P] = updateMeanAndCovWithSigmaPoints(x, P, y, s, h, R,
sigmaPoints, type)
%
%UPDATEGAUSSIANWITHSIGMAPOINTS computes the updated mean and covariance
%
%Input:
%   x           [n x 1] Prior mean
%   P           [n x n] Prior covariance
%   y           [m x 1] measurement
%   s           [2 x 1] sensor position
%   h           measurement model function handle
%   R           [m x m] measurement noise covariance matrix
%
%Output:
%   x           [n x 1] Updated mean
%   P           [n x n] Updated covariance
%
```

```
    % Compute sigma points
    [SP,W] = sigmaPoints(x, P, type);

    % Dimension of measurement
    m = size(R,1);

    % Dimension of state and number of sigma points
    [n, N] = size(SP);

    % Predicted measurement
    yhat = zeros(m,1);
    hSP = zeros(m,N);
    for i = 1:N
        [hSP(:,i),~] = h(SP(:,i),s);
        yhat = yhat + W(i)*hSP(:,i);
    end

    % Cross covariance and innovation covariance
    Pxy = zeros(n,m);
    S = R;
    for i=1:N
        Pxy = Pxy + W(i)*(SP(:,i)-x)*(hSP(:,i)-yhat)';
        S = S + W(i)*(hSP(:,i)-yhat)*(hSP(:,i)-yhat)';
    end

    % Ensure symmetry
    S = 0.5*(S+S');

    % Updated mean
    x = x+Pxy*(S\(y-yhat));
    P = P - Pxy*(S\(Pxy'));

    % Ensure symmetry
    P = 0.5*(P+P');
End
```

3.Resampling

```
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,N]=size(X);
[Xr,j]=datasample(X,N,2,'Weights',W');
Wr=ones(1,N)./N;
end
```

4.Particle filter update

```
function [X_k, W_k] = pfFilterStep(X_kmin1, W_kmin1, yk, 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


% Your code here!
X_k=mvnrnd(proc_f(X_kmin1)', proc_Q)';
```

```
W_k=W_kmin1.*mvnpdf(yk',meas_h(X_k)',meas_R)';
W_k=W_k./sum(W_k);
end
```

5. Particle filter

```
function [xfp, Pfp, Xp, Wp] = pfFilter(x_0, P_0, Y, proc_f, proc_Q, meas_h,
meas_R, ...
                             N, bResample, plotFunc)
%PFFILTER Filters measurements Y using the SIS or SIR algorithms and a
% state-space model.
%
% Input:
%   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] Non-resampled Particles for posterior state
distribution in times 1:K
%   Wp          [N x K] Non-resampled weights for posterior state x in times
1:K


% Your code here, please.
% If you want to be a bit fancy, then only store and output the particles if
the function
% is called with more than 2 output arguments.


    % Defining the dimensions
    n=size(x_0,1); [m,K]=size(Y);


    % Preallocates memory
    Xp=zeros(n,N,K); Wp=zeros(N,K);
```

```
    if nargout > 2  % If more than 2 output arguments
        xfp=zeros(n,K); Pfp=zeros(n,n,K);
    end

    %Non-resampled weigths for the posterior state x
    Wp(:,1)=repmat(1/N,[N,1]);

    %Particle for the posterior state from prior
    Xp(:,:,1)=mvnrnd(x_0,P_0,N)';

    %Particle for the posterior state from predicted
    Xp(:,:,1)=mvnrnd(proc_f(Xp(:,:,1))',proc_Q)';

    %Posterior means of the particle filter
    xfp(:,1)=Xp(:,:,1)*Wp(:,1);

    %Posterior error covariance of the particle filter
    Pfp(:,:,1)=(Xp(:,:,1)-xfp(:,1))*((Xp(:,:,1)-xfp(:,1))'.*Wp(:,1));

    figure();



    if bResample == true %filter with resampling
        Xr(:,:,1)=Xp(:,:,1); Wr(:,1)=Wp(:,1);
        for k=2:K+1
            [Xp(:,:,k),Wp(:,k)]=pfFilterStep(Xr(:,:,k-1),Wr(:,k-1)',Y(:,k-
1),proc_f,proc_Q,meas_h,meas_R);

            if nargout > 2  %more than 2 outputs
                xfp(:,k)=Xp(:,:,k)*Wp(:,k);
                Pfp(:,:,k)=(Xp(:,:,k)-xfp(:,k))*((Xp(:,:,k)-
xfp(:,k))'.*Wp(:,k));
            end
            [Xr(:,:,k),Wr(:,k),j]=resampl(Xp(:,:,k),Wp(:,k)');
            plotFunc(k,Xp(:,:,k),Xp(:,:,k-1),j);
        end
```

```matlab
    else %filter without resampling
        for k=2:K+1
            [Xp(:,:,k), Wp(:,k)] = pfFilterStep(Xp(:,:,k-1), Wp(:,k-1)', ...
                Y(:,k-1), proc_f, proc_Q, meas_h, meas_R);
            if nargout>2 %more than 2 outputs
                xfp(:,k)=Xp(:,:,k)*Wp(:,k);
                Pfp(:,:,k)=(Xp(:,:,k)-xfp(:,k))*((Xp(:,:,k)-
xfp(:,k))'.*Wp(:,k));
            end
            plotFunc(k,Xp(:,:,k),Xp(:,:,k-1),[1:1:N:N]);
        end
    end


    %deletion of first
    Xp=Xp(:,:,2:end); Wp=Wp(:,2:end);
    if nargout > 2
        xfp=xfp(:,2:end);
        Pfp=Pfp(:,:,2:end);
    end
end


function [Xk, Wk, j] = resampl(Xk, Wk)
    % Copy your code from previous task!
    [n,N]=size(Xk);
    [Xr,j]=datasample(Xk,N,2,'Weights',Wk');
    Wr=ones(1,N)./N;
end


function [X_k, W_k] = pfFilterStep(X_kmin1, W_kmin1, yk, proc_f, proc_Q,
meas_h, meas_R)
    % Copy your code from previous task!
    X_k=mvnrnd(proc_f(X_kmin1)',proc_Q)';
    W_k=W_kmin1.*mvnpdf(yk',meas_h(X_k)',meas_R)';

    W_k=W_k./sum(W_k);
end
```