TTK4250 - Sensor Fusion Assignment 3

Martin Eek Gerhardsen

27. September 2019

Department of Engineering Cybernetics

Contents

1 Task 1

1.1 a)

See appendix A.1 for implementation.

1.2 b)

See fig. 1, fig. 2, fig. 3 and fig. 4 for the results of using the script given with this assignment. Then, by comparing the resulting combinations to the original, we can conclude that:

- fig. 1b is best represented by combining 1 and 2.
- fig. 2b is best represented by combining 1 and 2.
- fig. 3b is best represented by combining 2 and 3.
- fig. 4b is best represented by combining 2 and 3.

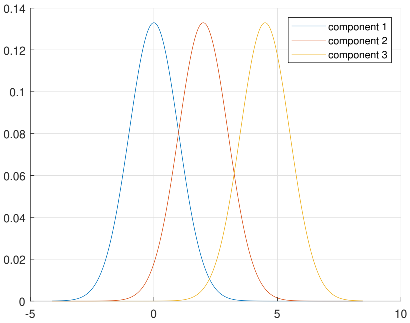
These combinations are mainly based on how close the mixed distributions are to the original distribution, but we can also motivate our choices by looking at the data. We'd like to loose as little information as possible when mixing, and therefore it would usually be best to look for this.

For i), the means are the only values which differs between the distributions, and we can see that most likely the two means closest are those of distribution 1 and 2.

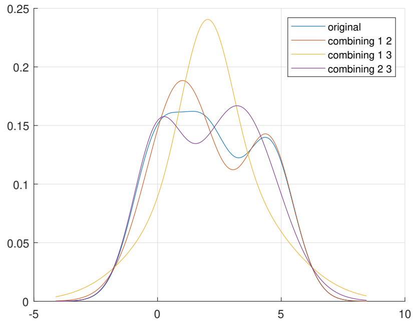
For ii) we can note that the weights are going to play a larger role in this distribution, and we can see that the original distribution, as well as most of the combinations are distributed around distribution 2. Here it would be dangerous to choose a mix without 2, and again as 1 and 2 have the closest means, this is our choice.

For iii) it is the variances which differs from earlier, and will *diffuse* their distributions. This means that these slopes will play a larger role in the mix as a whole, as 1 will only really affect the original in one area. Therefore, 2 and 3 give a good mix.

For iv) the problem is a bit more complicated, as two of the distributions have the same mean, however this also means that we should probably ignore one of these to get a better distribution. In our choice we do this, and combined with the argument from iii), we should conclude that 2 and 3 give the best mix.

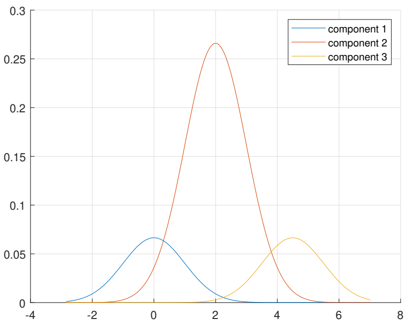


(a) The individual components

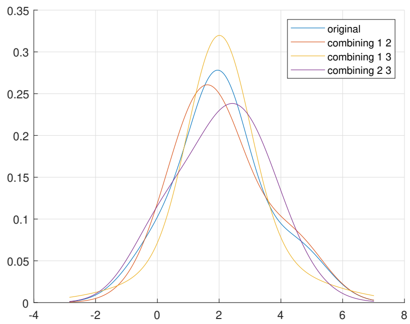


(b) The different combinations

Figure 1: i)

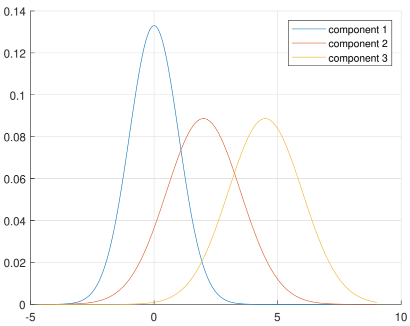


(a) The individual components

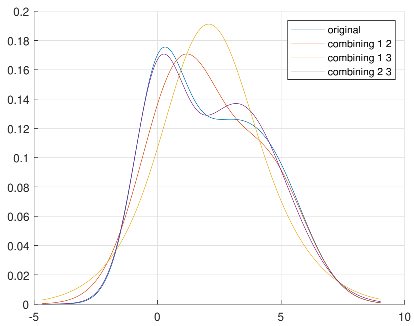


(b) The different combinations

Figure 2: ii)

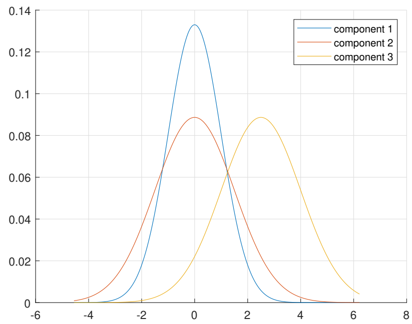


(a) The individual components

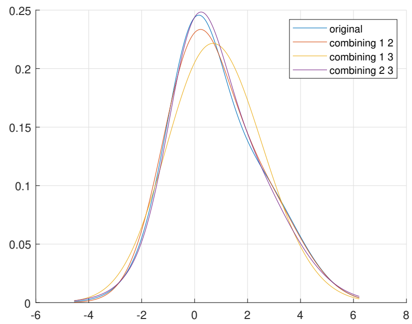


(b) The different combinations

Figure 3: iii)



(a) The individual components



(b) The different combinations

Figure 4: iv)

2 Task 2

Some useful equations:

$$\begin{aligned}
\Lambda_k^{(s_k)} &= p(\mathbf{z}_k|s_k, \mathbf{z}_{1:k-1}) && (1)\\
&= \int p(\mathbf{z}_k|x_k)p(x_k|s_k, \mathbf{z}_{1:k-1})dx_k && (2)\\
&= \mathcal{N}(\mathbf{z}_k, \mathbf{h}^{(s_k)}(\hat{\mathbf{x}}_{k|k-1}^{(s_k)}), \mathbf{S}_k^{(s_k)}) && (3)
\end{aligned}$$

2.1 a)

Using the equation from the assignment, we have:

$$p(z_k|z_{1:k-1}) = \sum_{s_k} \int p(z_k|x_k, s_k)p(x_k|s_k, z_{1:k-1})Pr(s_k|z_{1:k-1})dx_k \tag{4}$$

Then,

$$\begin{aligned}
p(z_k|z_{1:k-1}) &= \sum_{s_k} \int p(z_k|x_k, s_k)p(x_k|s_k, z_{1:k-1})Pr(s_k|z_{1:k-1})dx_k\\
&= \sum_{s_k} \int p(z_k|x_k)p(x_k|s_k, z_{1:k-1})Pr(s_k|z_{1:k-1})dx_k\\
&= \sum_{s_k} \Lambda_k^{(s_k)} Pr(s_k|z_{1:k-1})
\end{aligned}$$

2.2 b)

$$p(x_k|z_{1:k-1}) \approx \sum_{i=1}^{N} w_k^i \delta(x_k - x_k^i) \tag{5}$$

$$\begin{aligned}
p(z_k|z_{1:k-1}) &= \int p(z_k|x_k)p(x_k|z_{1:k-1})dx_k\\
&\approx \sum_{i=1}^{N} w_k^i \int p(z_k|x_k)\delta(x_k - x_k^i)dx_k = \sum_{i=1}^{N} w_k^i p(z_k|x_k^i)
\end{aligned}$$

3 Task 3

See appendix A.2 for implementation of all subtasks.

4 Task 4

4.1 Task 4a)

The values $r = 10$, $qCV = 4$ and $qCV = [10, 0.1]$ seemed to give decent results. The NIS-values seemed decently stable compared to the mean.

A MATLAB Code

A.1 Task 1a) Reduce Gauss Mix function

```
function [xmix, Pmix] = reduceGaussMix(w, x, P)
    % calculates the mean, xmix, and covariance, Pmix, of a mixture
    %   p(y) = sum_i w(i)*N(y; x(:, i), P(:, :, i))
    %
    % w (numel(w) x #mix): weights of the mixture
    % x (dim(state) x #mix): means of the mixture
    % P (dims(state) x dim(state) x #mix): covariances of the mixture
    %
    % xmix (dim(state) x #mix): total mean
    % Pmix (dim(state) x dim(state) x #mix): total covariance

    w = w(:);
    M = numel(w);
    n = size(x, 1);

    %% implementation
    % allocate
    xmix = zeros(n, 1);
    Pmix = zeros(n, n);

    % mean
    for i = 1:M
        xmix = xmix + w(i) * x(i);
    end

    % covariance
    P_inno = Pmix(:, :);
    for i = 1:M
        P_inno = P_inno + w(i) * x(:, i) * x(:, i)';
    end
    P_inno = P_inno - xmix * xmix';

    for i = 1:M
        Pmix = Pmix + w(i) * P(:, :, i);
    end
    Pmix = Pmix + P_inno;
end
```

A.2 Task 3) Implement an IMM class

```
classdef IMM
   properties
       modeFilters  % cell of EKFs
       PI           % markov transition matrix.
       M            % number of modes
   end
   methods
       function obj = IMM(modelcellarr, PI)
           % modelcell (M x 1 cell): cell array of EKFs
           % PI (M x M): Markov transition matrix
            obj = obj.setModel(modelcellarr, PI);
       end

       function obj = setModel(obj, modelcellarr, PI)
               % sets the internal functions and paramters
               %
               % modelcell (M x 1 cell): cell array of EKFs
               % PI (M x M): Markov transition matrix
               obj.modeFilters = modelcellarr;
               obj.PI = PI;
               obj.M = size(PI, 1);
       end

       function [spredprobs, smixprobs] = mixProbabilities(obj, sprobs)
           % IMM: step 1
           %
           % probs (M x 1): mode probabilities
           %
           % spredprobs (M x 1): predicted mode probabilities
           % smixprobs (M x M): mixing probabilities

           % Joint probability for this model and next
           spsjointprobs = obj.PI .* (ones(obj.M, 1) * sprobs(:)'); % ...

           % marginal probability for next model
           spredprobs = sum(spsjointprobs, 2); % ...

           % conditionional probability for model at this time step on th
           smixprobs = spsjointprobs ./ (spredprobs * ones(1, obj.M)); %
       end

       function [xmix, Pmix] = mixStates(obj, smixprobs, x, P)
```

```
            % IMM: step 2
            % smixprob (M x M): mixing probabilities
            % x (dim(state) x M): means to mix
            % P (dim(state) x dim(state) x M): covariances to mix
            %
            % xmix (dim(state) x M): mixed means
            % Pmix (dim(state) x dim(state) x M): mixed covariances

            % allocate
            xmix = zeros(size(x));
            Pmix = zeros(size(P));

            % mix for each mode,
            for i = 1:obj.M
                [xmix(:, i), Pmix(:, :, i)] = reduceGaussMix(smixprobs(i,
            end
        end

        function [xpred, Ppred] = modeMatchedPrediction(obj, x, P, Ts)
            % IMM: prediction part of step 3
            % x (dim(state) x M matrix): mean to predict
            % P (dim(state) x dim(state) x M): covariance to predict
            % Ts: sampling time for prediction.
            %
            % xpred (dim(state) x M): predicted means
            % Ppred (dim(state) x dim(state) x M): predicted covariances

            % allocate
            xpred = zeros(size(x));
            Ppred = zeros(size(P));

            % mode matched prediction
            for i = 1:obj.M
                [xpred(:, i), Ppred(:,:, i)] = obj.modeFilters{i}.predict(
            end
        end

        function [sprobspred, xpred, Ppred] = predict(obj, sprobs, x, P, T
            % IMM: step 1, 2 and prediction part of 3
            % sprobs (M x 1): mode probabilities
            % x (dim(state) x M): means to predict
            % P (dim(state) x dim(state) x M): covariances to predict
            % Ts: sampling time
            %
```

```
            % sprobspred (M x 1): predicted mode probabilities
            % xpred (dim(state) x M): predicted means
            % Ppred (dim(state) x dim(state) x M): predicted covariances

            % step 1
            [sprobspred, smixprobs] = obj.mixProbabilities(sprobs); % ...

            % step 2
            [xmix, Pmix] = obj.mixStates(smixprobs, x, P); % ...

            % prediction part of step 3
            [xpred, Ppred] = obj.modeMatchedPrediction(xmix, Pmix, Ts); %
        end

        function [xupd, Pupd, logLambdas] = modeMatchedUpdate(obj, z, x, P
            % IMM: update part of step 3
            % z (dim(measurement) x 1): measurement
            % x (dim(state) x M): the means to update
            % P (dim(state) x dim(state) x M): covariances to update
            %
            % xupd (dim(state) x M): updated means
            % Pupd (dim(state) x dim(state) x M): updated covariances
            % logLambdas (M x 1): measurement loglikelihood for given mode

            % allocate
            xupd = zeros(size(x));
            Pupd = zeros(size(P));
            logLambdas = zeros(obj.M, 1);

            % mode matched update and likelihood
            for i = 1:obj.M
                [xupd(:, i), Pupd(:,:, i)] = obj.modeFilters{i}.update(z,
                logLambdas = obj.modeFilters{i}.loglikelihood(z, x, P);
            end
        end

        function [supdprobs, loglikelihood] = updateProbabilities(obj, log
            % IMM: step 4
            %
            % logLambdas (M x 1): measurement loglikelihood for given mode
            % sprobs (M x 1): mode probabilities
            %
            % supdprobs (M x 1): updated mode probabilities
            % loglikelihood: measurement log likelilhood (total, ie. p(z_k
```

```

            % ... % you might want to do some precalculations here.
            logSporbs = log(sprobs);

            loglikelihood = logSumExp(logLambdas + logSporbs); % ... % you
            supdprobs = exp(logLambdas + logSporbs - loglikelihood); % ...
        end

        function [supdprobs, xupd, Pupd, loglikelihood] = update(obj, z, s
            % IMM: combining update part of step 3 and step 4
            %
            % z (dim(measurement) x 1): measurement
            % sprobs (M x 1): mode probabilities
            % x (dim(state) x M): the means to update
            % P (dim(state) x dim(state) x M): covariances to update
            %
            % supdprobs (M x 1): updated mode probabilities
            % xupd (dim(state) x M): updated means
            % Pupd (dim(state) x dim(state) x M): updated covariances
            % loglikelihood: measurement log likelilhood (total, ie. p(z_k

            % update part of step 3
            [xupd, Pupd, logLambdas] = obj.modeMatchedUpdate(z, x, P); % .

            % step 4
            [supdprobs, loglikelihood] = obj.updateProbabilities(logLambda
        end

        function [xest, Pest] = estimate(obj, sprobs, x, P)
            % IMM: step 5. A single mean and covariance as estimate. Reuse
            % of reduceGaussMix should simplify things.
            %
            % sprobs (M x 1): mode probabilities
            % x (dim(state) x M): means per mode
            % P (dim(state) x dim(state) x M): covariances per mode
            %
            % xest (dim(state) x M): MMSE/mean estimate
            % Pest (dim(state) x dim(state) x M): covariance of the estima

            [xest, Pest] = reduceGaussMix(sprobs, x, P);
        end

        function [NIS, NISes] = NIS(obj, z, sprobs, x, P)
            % calculate the NIS for each mode, and one for the averaged
```

```
                % innvoations.
                %
                % sprobs (M x 1): mode probabilities
                % x (dim(state) x M): means per mode
                % P (dim(state) x dim(state) x M): covariances per mode
                %
                % NIS (scalar): NIS calculated based on the estimation mean an
                % NISes (M x 1): NIS for each mode

                m = size(z,1);
                NISes = zeros(obj.M, 1);
                innovs = zeros(m, obj.M);
                Ss = zeros(m, m, obj.M);
                for s = 1:obj.M
                    [innovs(:, s), Ss(:, :, s)] = obj.modeFilters{s}.innovatio
                    NISes(s) = obj.modeFilters{s}.NIS(z, x(:, s), P(:,:,s));
                end
                [totInnov, totS] = reduceGaussMix(sprobs, innovs, Ss);
                NIS = totInnov' * (totS \ totInnov);
            end
        end
    end

function lse = logSumExp(a)
    % more numerically stable way(less chance of underflow and overflow)
    % to calculate logsumexp of a list, a.
    %
    % uses the fact
    % log(sum(exp(a))) = log(sum(exp(b)exp(a - b))
    % = log(exp(b)sum(exp(a - b))) = b + log(sum(exp(a - b)))
    % where we let b = max(a),
    amax = max(a(:));
    lse = amax + log(sum(exp(a - amax)));
end
```

A.3 Task 4) Tune an IMM

```
% load data
usePregen = true; % you can generate your own data if set to false
if usePregen
    load task4data.mat;
else
    K = 100;
```

```
    Ts = 2.5;
    r = 5;
    q = [0.005, 1e-6*pi]; % q(1): CV noise (effective all the time), q(2)
    init.x = [0; 0; 2; 0; 0];
    init.P = diag([25, 25, 3, 3, 0.0005].^2);
    [Xgt, Z] = simulate_atc(q, r, K, init, false);
end


figure(1); clf; hold on; grid on;
plot(Xgt(1,:), Xgt(2,:));
scatter(Z(1,:), Z(2, :));


%%
% tune single filters
r = 10;
qCV = 4;
qCT = [10, 0.1];

% choose model to tune
s = 2;

% make models
models =  cell(2,1);
models{1} = EKF(discreteCVmodel(qCV, r));
models{2} = EKF(discreteCTmodel(qCT, r));

% % % % allocate
xbar = zeros(5, 100);
Pbar = zeros(5, 5, 100);
xhat = zeros(5, 100);
Phat = zeros(5, 5, 100);
NIS = zeros(100, 1);

% initialize filter
xbar(:, 1) = [0; 0; 2; 0; 0];
Pbar(:, : , 1) = diag([25, 25, 3, 3, 0.0005].^2);

% filter
for k = 1:K
    [xhat(:, k), Phat(:, :, k)] = models{s}.update(Z(:, k), xbar(:, k), P
    NIS(k) = models{s}.NIS(Z(:, k), xbar(:, k), Pbar(:, :, k));
    if k < K
```

```
        [xbar(:, k + 1), Pbar(:, :, k + 1)] = models{s}.predict(xhat(:, k
    end
end

% errors
poserr = sqrt(sum((xhat(1:2,:) - Xgt(1:2,:)).^2, 1));
% posRMSE =
velerr = sqrt(sum((xhat(3:4, :) - Xgt(3:4, :)).^2, 1));
% velRMSE =

% consistency
confidenceInterval = chi2inv([0.05, 0.95], 2 * K) / K;
ANIS = mean(NIS)

% plot
figure(2); clf; hold on; grid on;
plot(xhat(1,:), xhat(2,:));
scatter(Z(1,:), Z(2, :));
% title(sprintf('posRMSE = %.3f, velRMSE = %.3f',posRMSE, velRMSE))

figure(3); clf; hold on; grid on;
plot(xhat(5,:))
% plot(Xgt(5,:))
ylabel('omega')

figure(4); clf;
% subplot(3,1,1)
plot(1:K, NIS); grid on;
ylabel('NIS')
% subplot(3,1,2);
% plot(poserr); grid on;
% ylabel('pos error')
% subplot(3,1,3)
% plot(velerr); grid on;
% ylabel('vel error')
%%
% tune IMM by only looking at the measurements
r = 1;
qCV = 1;
qCT = [1, 1];
p11 = 0.9;
p22 = 0.9;
PI = [%..., %...; %..., %...];
assert(all(sum(PI, 1) == [1, 1]),'columns of PI must sum to 1')
```

```
% make model
models =  cell(2,1);
models{1} = EKF(discreteCVmodel(qCV, r));
models{2} = EKF(discreteCTmodel(qCT, r));
imm = IMM(models, PI);

% allocate
xbar = zeros(5, 2, K); % dims: state, models, time
Pbar = zeros(5, 5, 2, K); % dims: state, state, models, time
probbar = zeros(2, K);
xhat = zeros(5, 2, K);
xest = zeros(5, K);
Pest = zeros(5, 5, K);
Phat = zeros(5, 5, 2, K);
probhat = zeros(2, K);
NIS = zeros(K, 1);
NISes = zeros(2, K);

% initialize
xbar(:, :, 1) = repmat(%..., [1, 2]);
Pbar(:, : ,:, 1) = repmat(%...,[1,1,2]);
probbar(:, 1) = [%...; %...];

% filter
for k=1:100
    [NIS(k), NISes(:, k)] = imm.NIS(Z(:, k), sprobs, x, P);
    [probhat(:, k), xhat(:, :, k), Phat(:, :, :, k)] = ...
        %...
    [xest(:, k), Pest(:, :, k)] = %...
    if k < 100
        [probbar(:, k+1), xbar(:, :, k+1), Pbar(:, :, :, k+1)] = ...
            %...
    end
end

% consistency
confidenceInterval = % ...
ANIS = mean(NIS)

% plot
figure(5); clf; hold on; grid on;
plot(xest(1,:), xest(2,:));
scatter(Z(1,:), Z(2, :));
```

```

figure(6); clf; hold on; grid on;
plot(xest(5,:))
ylabel('omega')

figure(7); clf;
plot(probhat');
grid on;
ylabel('Pr(s)')

figure(8); clf; hold on; grid on;
plot(NIS)
plot(NISes')
ylabel('NIS')
%%
% % tune IMM by looking at ground truth
% r = %...;
% qCV = %...;
% qCT = [%..., %...];
% PI = [%...,%...; %..., %...];
% assert(all(sum(PI, 1) == [1, 1]),'columns of PI must sum to 1')
%
% % make model
% models =  cell(2,1);
% models{1} = EKF(discreteCVmodel(qCV, r));
% models{2} = EKF(discreteCTmodel(qCT, r));
% imm = IMM(models, PI);
%
% % allocate
% xbar = zeros(5, 2, K);
% Pbar = zeros(5, 5, 2, K);
% probbar = zeros(2, K);
% xhat = zeros(5, 2, K);
% xest = zeros(5, K);
% Pest = zeros(5, 5, K);
% Phat = zeros(5, 5, 2, K);
% probhat = zeros(2, K);
% NIS = zeros(K, 1);
% NISes = zeros(2, K);
% NEES = zeros(K, 1);
%
% % initialize
% xbar(:, :, 1) = repmat(%..., [1, 2]);
% Pbar(:, : ,:, 1) = repmat(%...,[1,1,2]);
```

```
% probbar(:, 1) = [%...; %...];
%
% % filter
% for k=1:100
%     [NIS(k), NISes(:, k)] = %...
%
%     [probhat(:, k), xhat(:, :, k), Phat(:, :, :, k)] = %...
%
%     [xest(:, k), Pest(:, :, k)] = %...
%
%     NEES(k) = %...
%     if k < 100
%         [probbar(:, k+1), xbar(:, :, k+1), Pbar(:, :, :, k+1)] = %...
%     end
% end
%
% % errors
% poserr = sqrt(sum((xest(1:2,:) - Xgt(1:2,:)).^2, 1));
% posRMSE = %... % not true RMSE (which is over monte carlo simulations)
% velerr = sqrt(sum((xest(3:4, :) - Xgt(3:4, :)).^2, 1));
% velRMSE = %... % not true RMSE (which is over monte carlo simulations)
% % peakPosDeviation =
% % peakVelDeviation =
%
% % consistency
% confidenceIntervalNIS = % ...
% ANIS = mean(NIS)
% confidenceIntervalNEES = % ...
% ANEES = mean(NEES)
%
% % plot
% figure(9); clf; hold on; grid on;
% plot(xest(1,:), xest(2,:));
% plot(Xgt(1,:), Xgt(2, :));
% title(sprintf('posRMSE = %.3f, velRMSE = %.3f',posRMSE, velRMSE))
%
% figure(10); clf; hold on; grid on;
% plot(xest(5,:))
% plot(Xgt(5,:))
%
% figure(11); clf;
% plot(probhat');
% grid on;
%
```

```
% figure(12); clf;
% subplot(4,1,1);
% plot(poserr); grid on;
% ylabel('position error')
% subplot(4,1,2);
% plot(velerr); grid on;
% ylabel('velocity error')
% subplot(4,1,3); hold on; grid on;
% plot(NIS)
% plot(NISes')
% ciNIS = chi2inv([0.05, 0.95], 2);
% inCI = sum((NIS >= ciNIS(1)) .* (NIS <= ciNIS(2)))/K;
% plot([1,K], repmat(ciNIS',[1,2])','r--')
% text(104, -2, sprintf('%.2f%% inside CI', inCI),'Rotation',90);
% ylabel('NIS');
% subplot(4,1,4);
% plot(NEES); grid on; hold on;
% ylabel('NEES');
% ciNEES = chi2inv([0.05, 0.95], 4);
% inCI = sum((NIS >= ciNEES(1)) .* (NIS <= ciNEES(2)))/K;
% plot([1,K], repmat(ciNEES',[1,2])','r--')
% text(104, -5, sprintf('%.2f%% inside CI', inCI),'Rotation',90);
```