Fall 2024 – 16-642 Manipulation, Estimation, and Control
Problem Set 3

Ankit Aggarwal (ankitagg)

Question 1

(a) Linearised System Derivation is shown below.

Ques. 1

(a) Linearized Approximation

$$q[k+1] = \begin{bmatrix} q_1[k] + T(u_1[k] + v_1[k]) \cos q_3[k] \\ q_2[k] + T(u_1[k] + v_1[k]) \sin q_3[k] \\ q_3[k] + T(u_2[k] + v_2[k]) \end{bmatrix}$$

To find linearized approximation,

$$q[k+1] \simeq F(q[k], u[k]) \cdot q[k] + G(q[k]) \cdot u[k] + \Gamma(q[k]) \cdot v[k]$$

$$F(q[k], u[k]) = \left.\frac{dq(k+1)}{dq}\right|_{q = \hat{q}[k|k],\, u = u[k]}$$

$$F = \begin{bmatrix} \frac{\partial f_1}{\partial q_1} & \cdots & \frac{\partial f_1}{\partial q_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial q_1} & \cdots & \frac{\partial f_n}{\partial q_n} \end{bmatrix}$$

$$F = \begin{bmatrix} 1 & 0 & -T\left(u_1[k]+v_1[k]\right)\sin\left(q_3[k]\right) \\ 0 & 1 & T\left(u_1[k]+v_1[k]\right)\cos\left(q_3[k]\right) \\ 0 & 0 & 1 \end{bmatrix}$$

$$G(q[k]) = \frac{\partial q[k+1]}{\partial u}\bigg|_{q=\hat{q}[k|k],\, u=u[k]}$$

$$G = \begin{bmatrix} T\cos(q_3[k]) & 0 \\ T\sin(q_3[k]) & 0 \\ 0 & T \end{bmatrix}$$

$$\Gamma(q[k]) = \frac{\partial q[k+1]}{\partial v}\bigg|_{\substack{q=\hat{q}[k|k], \\ v=v[k]}}$$

$$\Gamma(q[k]) = \begin{bmatrix} T\cos(q_3[k]) & 0 \\ T\sin(q_3[k]) & 0 \\ 0 & T \end{bmatrix}$$

(b) To find the noise covariances, I used the given system equations and derived expressions for each component of the noises.
Starting with measurement noise **w[k],** the equation is shown below:

For measurement noise w[k],

$$w[k] = y[k] - \begin{bmatrix} q_1[k] \\ q_2[k] \end{bmatrix}$$

To find the required data, I modelled the equations in MATLAB:

```
load("calibration.mat")
T = 0.01;
q_trim = trimdata(q_groundTruth, [2, 2501]);
q_data = zeros(2, 250);
w_data = zeros(2, 250);


%Finding covariance of Measurement Noise
for i = 1:length(t_y)
    index = round(t_y(i) / T) + 1;
    if index > 0 && index <= size(q_trim, 2)
        q_data(:, i) = q_trim(:, index);
        w_data(:, i) = y(:, i) - q_data(:, i);
    end
end


W = cov(w_data')
```

This code samples the given true state from q_groundTruth at the corresponding times where y is available (i.e., every 10 timesteps). This sampled q_data is then subtracted from y to obtain the Measurement Noise (w_data). Further, the cov function is used to find the required 2x2 covariance matrix.

```
W = 2×2
      1.8817    0.0632
      0.0632    2.1384
```

To find the process noise v[k], the equation is shown below:

For process noise $v[k]$,
Using the equation for the first element of q,

$$q_1[k+1] = q_1[k] + T\left(u_1[k] + v_1[k]\right)\cos q_3[k]$$

$$\Rightarrow v_1[k] = \frac{q_1[k+1] - q_1[k]}{T \cos q_3[k]} - u_1[k]$$

Using the equation for the third element of q,

$$q_3[k+1] = q_3[k] + T\left(u_2[k] + v_2[k]\right)$$

$$\Rightarrow v_2[k] = \frac{q_3[k+1] - q_3[k]}{T}$$

To find the required data, I modelled the equations in MATLAB:

```
% Finding covariance of Process Noise
v_data = zeros(2,length(q_groundtruth));


for j = 1:length(q_groundtruth)-1
    v_data(1,j) = ((q_groundtruth(1,j+1) - q_groundtruth(1,j)) / (T *
cos(q_groundtruth(3,j)))) - u(1,j);
    v_data(2,j) = (q_groundtruth(3,j+1) - q_groundtruth(3,j)) / T - u(2,j);
end
V = cov(v_data')
```

This code uses the derived equations to find the two elements of process noise (v1 and v2) and stores them in the v_data matrix. Further, the cov function is used to find the required 2x2 covariance matrix.

```
V = 2x2
      0.2590    0.0010
      0.0010    0.0625
```

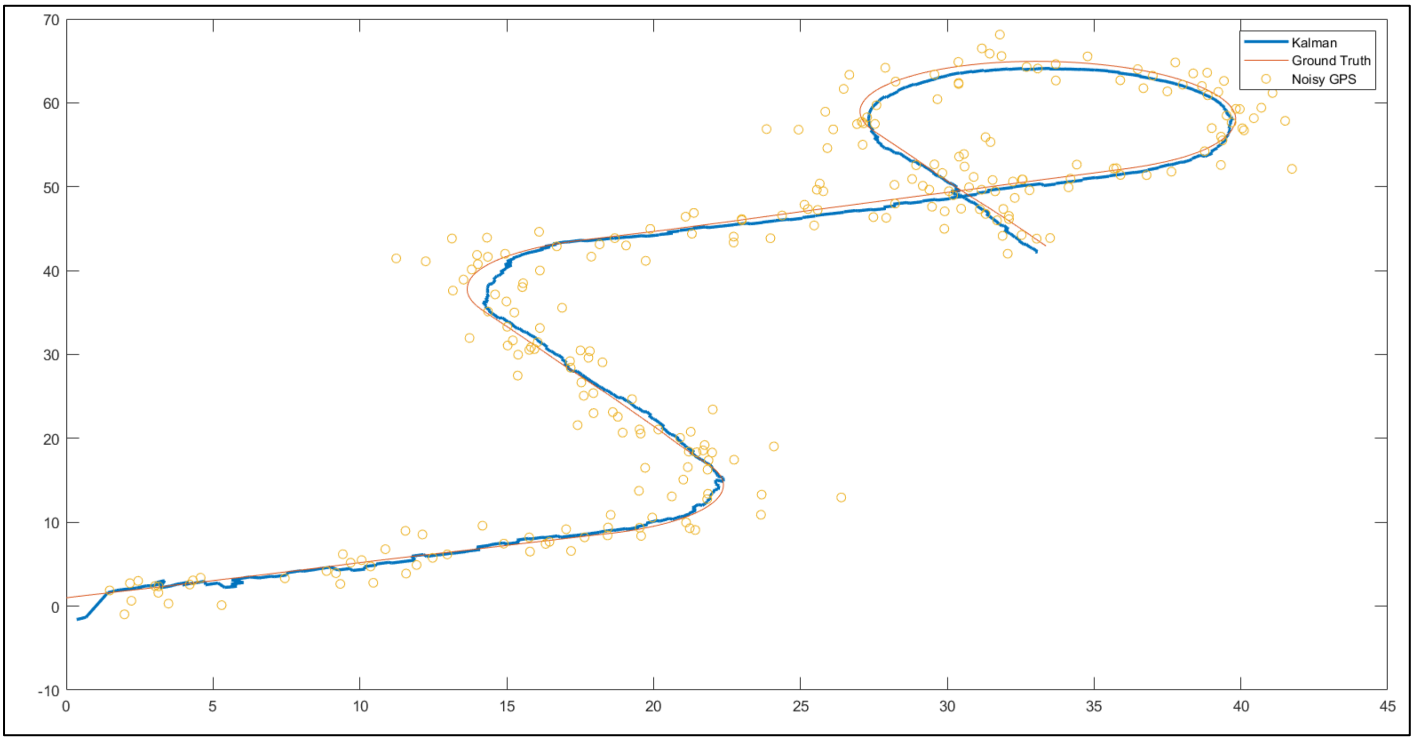
(c) Using the extended Kalman Filter concepts, the code I used is shown below,

```
clear;

load("calibration.mat")
load("kfData.mat")
T = 0.01;
q_trim = trimdata(q_groundtruth, [2, 2501]);
q_data = zeros(2, 250);
w_data = zeros(2, 250);
%Finding covariance of Measurement Noise
```

```
for i = 1:length(t_y)
    index = round(t_y(i) / T) + 1;
    if index > 0 && index <= size(q_trim, 2)
        q_data(:, i) = q_trim(:, index);
        w_data(:, i) = y(:, i) - q_data(:, i);
    end
end
W = cov(w_data')
% Finding covariance of Process Noise
v_data = zeros(2,length(q_groundtruth));
for j = 1:length(q_groundtruth)-1
    v_data(1,j) = ((q_groundtruth(1,j+1) - q_groundtruth(1,j)) / (T *
cos(q_groundtruth(3,j)))) - u(1,j);
    v_data(2,j) = (q_groundtruth(3,j+1) - q_groundtruth(3,j)) / T - u(2,j);
end
V = cov(v_data')

%Generating random noise
W_R = chol(W, 'lower');
w_noise = (repmat([0 0],length(q_groundtruth),1) + randn(length(q_groundtruth),2)*W_R)';
V_R = chol(V, 'lower');
v_noise = (repmat([0 0],length(q_groundtruth),1) + randn(length(q_groundtruth),2)*V_R)';

% System Definition
F = @(x,u,v)[x(1)+T*(u(1)+v(1))*cos(x(3));
         x(2)+T*(u(1)+v(1))*sin(x(3));
         x(3)+T*(u(2)+v(2))];
H = @(x,w)[x(1)+w(1);
         x(2)+w(2)];
state_initial = [0.355 -1.590 0.682]';
P  = [25 0 0; 0 25 0; 0 0 0.154];

x_estimate = zeros(3,length(q_groundtruth)-1);x_estimate(:,1) = state_initial;
covariances = zeros(3,3,length(x_estimate)); covariances(:,:,1) = P;
%EKF Algorithm
for k = 2:2501

    %prediction step
    x_estimate(:,k) = F(x_estimate(:,k-1),u(:,k-1),v_noise(:,k-1));
    F_l = [1 0 0; 0 1 0; -T*(u(1,k-1)+v_noise(1,k-1))*sin(x_estimate(3,k-1)) T*(u(1,k-
1)+v_noise(1,k-1))*cos(x_estimate(3,k-1)) 1]'; %recheck
    G_l = [T*cos(x_estimate(3,k-1)) 0; T*sin(x_estimate(3,k-1)) 0; 0 T];
    Gamma_l = [T*cos(x_estimate(3,k-1)) 0; T*sin(x_estimate(3,k-1)) 0; 0 T];
    H_l = [1 0 0; 0 1 0];
    covariances(:,:,k) = F_l * covariances(:,:,k-1) * F_l' + Gamma_l * V * Gamma_l';
    K = covariances(:,:,k) * H_l' * inv(H_l*covariances(:,:,k)*H_l' + W);


    % update when given measurement
    if mod(k,10) == 0
       element_index = k/10;
       x_estimate(:,k) = x_estimate(:,k) +  K*(y(:,element_index) - H(x_estimate(:,k),[0
0]));
       covariances(:,:,k) = (eye(3) - K*H_l) * covariances(:,:,k);
    end
end
% Plotting Results
figure()
```

```
plot(x_estimate(1,:), x_estimate(2,:), 'LineWidth',2)
hold on
plot(q_groundtruth(1,:),q_groundtruth(2,:))
hold on
scatter(y(1,:), y(2,:));
hold on


legend('Kalman', 'Ground Truth', 'Noisy GPS')
```

The final plot obtained is shown below:



Question 2

To implement the particle filter, I wrote the following code using the provided pfTemplate.m code structure:

```
function M = pfTemplate()
% template and helper functions for 16-642 PS3 problem 2

rng(0); % initialize random number generator

b1 = [5,5]; % position of beacon 1
b2 = [15,5]; % position of beacon 2

load("pfData.mat");
% initialize movie array
numSteps = length(t);
M(numSteps) = struct('cdata',[],'colormap',[]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%         put particle filter initialization code here                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
T = t(2) - t(1); % T = 0.1
```

I chose the timestep as 0.1 by referring to the ‘t’ variable in the pfData.mat provided.

```
% System Definition
F = @(x,u,v)[x(1)+T*(u(1)+v(1))*cos(x(3));
             x(2)+T*(u(1)+v(1))*sin(x(3));
             x(3)+T*(u(2)+v(2))];

H = @(x,w)[sqrt((x(1)-b1(1))^2+(x(2)-b1(2))^2)+w(1);
           sqrt((x(1)-b2(1))^2+(x(2)-b2(2))^2)+w(2)];


numParticles = 1200;
x_min = 0; x_max = 20; y_min = 0; y_max = 10; theta_min = 0; theta_max = 2*pi;
particles = [x_min+(x_max-x_min)*rand(1,numParticles); y_min+(y_max-
y_min)*rand(1,numParticles); theta_min+(theta_max-theta_min)*rand(1,numParticles)];
%disp(particles)
```

I chose the number of particles based on several tuning runs until I got to my best result. The bounds for x and y were chosen based on the axes given in the helper function and the bounds for theta based on one complete 360-degree rotation. Using these bounds, I was able to create a uniform distribution of particles that serves as my initial estimate.

```

% here is some code to plot the initial scene
figure(1)
plotParticles(particles); % particle cloud plotting helper function
hold on
plot([b1(1),b2(1)],[b1(2),b2(2)],'s',...
    'LineWidth',2,...
    'MarkerSize',10,...
    'MarkerEdgeColor','r',...
    'MarkerFaceColor',[0.5,0.5,0.5]);
drawRobot(q_groundTruth(:,1), 'cyan'); % robot drawing helper function
plot(q_groundTruth(1,:), q_groundTruth(2,:));
axis equal
axis([0 20 0 10])
M(1) = getframe; % capture current view as movie frame
pause
disp('hit return to continue')

W = eye(2) * 0.7;
V = eye(2) * 0.7;
```

I chose the number of particles based on several tuning runs until I got to my best result. My initial chosen values were identity matrices. The intuition used to tune depended on analysing the error seen in the output video and trying to reason which element of noise could be affecting the state estimate.

```
% iterate particle filter in this loop
for k = 2:numSteps
    %Generating random noise
    v_generated = mvnrnd([0, 0], V, numParticles);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %               put particle filter prediction step here               %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    predictedMeasure = zeros(2,length(particles));
    for m = 1:numParticles
        particles(:,m) = F(particles(:,m),u(:,k-1),v_generated(m,:)');
        particles(1,m) = max(x_min, min(x_max, particles(1,m)));
        particles(2,m) = max(y_min, min(y_max, particles(2,m)));
        particles(3,m) = mod(particles(3,m), 2*pi);
        predictedMeasure(:,m) = H(particles(:,m), [0 0]);
    end
```

This code generates random noise based on the normal distribution with zero-mean and tuned covariance V. The filter prediction step entails moving the particles based on the system's defined motion model and predicting an output based on the new state estimate. I also included lines of code to ensure that the particles do not leave the defined bounds of x and y to minimize particle loss and stray values.

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %                 put particle filter update step here                 %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % weight particles
    weights = zeros(1, numParticles);
    for index = 1:length(weights)
        weights(index) = (1/(2*pi * sqrt(det(W)))) * exp(-0.5*(y(:,k)-
predictedMeasure(:,index))' * inv(W) * (y(:,k)-predictedMeasure(:,index)));
    end
    weights = weights / sum(weights);
```

To calculate weights, I used the multivariate gaussian distribution equation shown below:

$$\boxed{p(x;\mu,\Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)}$$

where, $n = 2$, $x = y$, $\mu =$ predictedMeasure, $\Sigma = W$

This equation uses the noisy GPS measurements and assigns weights to each particle based on how close they are to the GPS measurement. This means that particles closer to the measured output will be assigned a higher weight. After calculating each weight, I normalised it by dividing each weight by the sum of all weights.

```
    % resample particles
    CumulativeWeights = cumsum(weights);
```

```
    newParticles = zeros(size(particles));

    for g = 1:length(newParticles)
        q = rand();
        jdx = find(CumulativeWeights > rand, 1);
        newParticles(:,g) = particles(:,jdx);
    end
    particles = newParticles;

    robot_est = mean(particles,2);
```

The resampling steps requires sampling with replacement of particles based on the assigned weight to each particle.

To do this, I calculated the cumulative sum of weights and stored it in the CumulativeWeights array (*Inverse Transform Sampling Technique*) . As each particle's weight lies between the 0,1 and the total sum will be equal to 1, I can use the rand function for sampling. When the rand function gives a number between 0 and 1, I find the first index where the CumulativeWeight > rand. This is weighted sampling as the elements with higher weights will have a larger segment of [0,1], effectively equal to having more of a chance of being sampled.

These re-sampled particles are stored in newParticles. After the loop ends, the particles variable is updated. I waited until the loop ends to update to ensure that the sampling occurs from the original particles only and the new ones do not affect the resampling process.

To estimate the robot state based on these particles, I find the mean state of all particles.

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % plot particle cloud, robot, robot estimate, and robot trajectory here %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    figure(k)
    plot(q_groundTruth(1,:), q_groundTruth(2,:));
    hold on
    plot([b1(1),b2(1)],[b1(2),b2(2)],'s',...
    'LineWidth',2,...
    'MarkerSize',10,...
    'MarkerEdgeColor','r',...b
    'MarkerFaceColor',[0.5,0.5,0.5]);
    plotParticles(particles);
    drawRobot(robot_est, 'cyan'); % robot drawing helper function
    drawRobot(q_groundTruth(:,k), 'magenta')
    axis equal
    axis([0 20 0 10])
    legend('Ground Truth Trajectory','Beacons', '', 'Particles', '', 'Robot Estimate', '',
'Robot Ground Truth')

    % capture current figure and pause
    M(k) = getframe; % capture current view as movie frame
    pause
    disp('hit return to continue')

end


% when you're ready, the following block of code will export the created
% movie to an mp4 file
videoOut = VideoWriter('result.mp4','MPEG-4');
```

```
videoOut.FrameRate=5;
open(videoOut);
for k=1:numSteps
  writeVideo(videoOut,M(k));
end
close(videoOut);


close all;
end


% helper function to plot a particle cloud
function plotParticles(particles)
plot(particles(1, :), particles(2, :), 'go')
line_length = 0.1;
quiver(particles(1, :), particles(2, :), line_length * cos(particles(3, :)), line_length *
sin(particles(3, :)))
end

% helper function to plot a differential drive robot
function drawRobot(pose, color)

% draws a SE2 robot at pose
x = pose(1);
y = pose(2);
th = pose(3);

% define robot shape
robot = [-1 .5 1 .5 -1 -1;
          1  1 0 -1  -1 1 ];
tmp = size(robot);
numPts = tmp(2);
% scale robot if desired
scale = 0.5;
robot = robot*scale;

% convert pose into SE2 matrix
H = [ cos(th)   -sin(th)  x;
      sin(th)    cos(th)  y;
      0          0        1];

% create robot in position
robotPose = H*[robot; ones(1,numPts)];

% plot robot
plot(robotPose(1,:),robotPose(2,:),'k','LineWidth',2);
rFill = fill(robotPose(1,:),robotPose(2,:), color);
alpha(rFill,.2); % make fill semi transparent
end
```

The final part of the code includes plotting functions and all helper functions given to us.

The final output is the result.mp4 video submitted alongside this writeup which includes the beacon locations, robot ground truth pose and trajectory, particle cloud and the robot pose estimate derived from the particles.

Question 3

Q3 Autonomous Underwater Robot

Hazardous Structures (H)

$P(H) = 0.35$, $P(H') = 0.65$

Strong Currents (C)

$P(SC) = 0.15$, $P(SC') = 0.85$

Sonar Sensor (SS):

$P(SS=1 \mid H=1) = 0.9$

$P(SS=0 \mid H=1) = (1-0.9) = 0.1$

$P(SS=1 \mid H=0) = 0.02$

$P(SS=0 \mid H=0) = (1-0.02) = 0.98$

Current Flow Sensor (CF)

$P(CF=1 \mid SC=1) = 0.45$

$P(CF=0 \mid SC=1) = (1-0.45) = 0.55$

$P(CF=1 \mid SC=0) = 0.04$

$P(CF=0 \mid SC=0) = (1-0.04) = 0.96$

Part (a) –

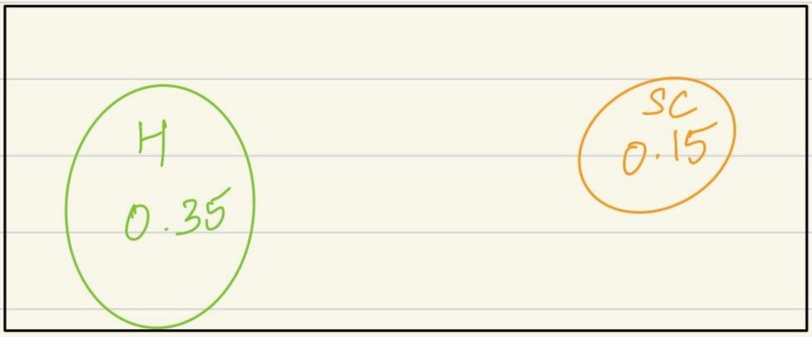
(a) Finding the probability of the robot encountering a strong current given a warning from either sensor

Let the event of getting a warning = W

To find: $P(SC \mid W)$

$$= \frac{P(W \mid SC)\ P(SC)}{P(W)}$$

To find $P(W)$,
we know that SC and H are disjoint events, the sample space can be represented as.



$$P(W) = P(SS \cup CF)$$

$$= P(SS) + P(CF) - P(SS \cap CF)$$

To find $P(SS)$,

$$P(SS) = P(SS|H)\,P(H) + P(SS|H')\,P(H')$$

$$= 0.9 \times 0.35 + 0.02 \times 0.65$$

$$P(SS) = 0.328$$

To find $P(CF)$,

$$P(CF) = P(CF|SC)\,P(SC) + P(CF|SC')\,P(SC')$$

$$= 0.45 \times 0.15 + 0.04 \times 0.85$$

$$P(CF) = 0.1015$$

So,

$$P(W) = 0.328 + 0.1015 - (0.1015)(0.328)$$

$$= 0.396208$$

Now, to find $P(W|SC)$

$$P(W|C) = P(SS \cup CF \,|SC)$$

$$P(W|C) = P(SS|SC) + P(CF|SC) - P(SS|SC) \cdot P(CF|SC)$$

Since the Sonar Sensor does not detect strong currents, the probability

$$P(SS|SC) = P(SS|H')$$

$$\Rightarrow P(W|C) = P(SS|H') + P(CF|SC) - P(SS|H')\, P(CF|SC)$$
$$= 0.02 + 0.45 - 0.02 \times 0.45$$
$$= 0.461$$

Hence,

$$P(SC|W) = \frac{0.461 \times 0.15}{0.396208}$$

$$\boxed{P(SC|W) = 0.17452}$$

Part (b) –

(b) How many consecutive measurements from the CF sensor need to be taken to get 0.9 probability?

Given that $P(CF=1 \mid SC=1) = 0.45$

The probability of correct identification of a strong current

$P(\text{correct identification})$
$= 1 - P(\text{Incorrect Identification})$

Propagating this to 'n' measurements,

$P(\text{correct identification})$
$= 1 - P(\text{Incorrect Identification})^n$
$= 1 - (1 - 0.45)^n$
$= 1 - (0.55)^n$

Target probability $= 0.9$

Equating,

$$1-(0.55)^n \geq 0.9$$

$$\Rightarrow (0.55)^n \geq 0.1$$

$$\Rightarrow n \log(0.55) \geq \log(0.1)$$

$$\Rightarrow n \geq \frac{\log(0.1)}{\log(0.55)} \sim 3.85$$

$$\boxed{n=4} \quad \text{rounding up}$$

Hence, 4 consecutive measurements are needed from the current flow sensor to get the same probability of correctly identifying an imminent danger to the robot.

Part (c) –

(c) Find variance σ^2 such that when averaged over 'n' measurements, it achieves the same aggregate noise.

$n=4$

Sonar Sensor Noise $= N(0,7)$

when taking multiple independent measurements, the variance of the average decreases.

Variance of average over 'n' measurements $= \frac{\sigma^2}{n}$

We want the σ^2 to be 7 after $n=4$ measurements.

$\Rightarrow \frac{\sigma^2}{4} = 7 \Rightarrow \sigma^2 = 4 \cdot 7$

$\Rightarrow \boxed{\sigma^2 = 28}$