Question 1.a

Monday, November 14, 2022 12:25 AM

(a) (10 points) Find the linearized approximation of the system as a function of the current state. In particular, your approximation should look like

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

where $F(q[k], u[k])$ is a 3×3 matrix, and $G(q[k])$ and $\Gamma(q[k])$ are both 3×2.

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

$$F = \frac{df}{dq}\bigg|_{V=0}$$

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

$$G = \frac{df}{du}\bigg|_{V=0}$$

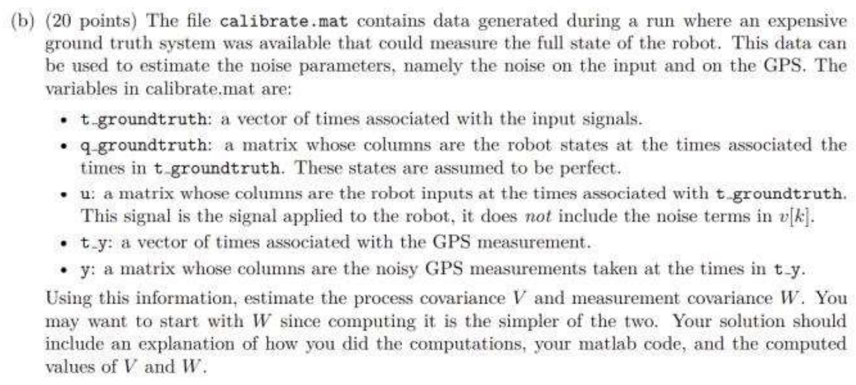
$$G = \begin{bmatrix} T\cos q_3[k] & 0 \\ T\sin q_3[k] & 0 \\ 0 & T \end{bmatrix}$$

$$\Gamma = \frac{\partial f}{\partial v}\bigg|_{V=0}$$

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

Question 1.b

Monday, November 14, 2022 12:45 AM



(b) (20 points) The file `calibrate.mat` contains data generated during a run where an expensive ground truth system was available that could measure the full state of the robot. This data can be used to estimate the noise parameters, namely the noise on the input and on the GPS. The variables in calibrate.mat are:

- `t_groundtruth`: a vector of times associated with the input signals.
- `q_groundtruth`: a matrix whose columns are the robot states at the times associated the times in `t_groundtruth`. These states are assumed to be perfect.
- `u`: a matrix whose columns are the robot inputs at the times associated with `t_groundtruth`. This signal is the signal applied to the robot, it does *not* include the noise terms in $v[k]$.
- `t_y`: a vector of times associated with the GPS measurement.
- `y`: a matrix whose columns are the noisy GPS measurements taken at the times in `t_y`.

Using this information, estimate the process covariance V and measurement covariance W. You may want to start with W since computing it is the simpler of the two. Your solution should include an explanation of how you did the computations, your matlab code, and the computed values of V and W.

Computed Values of V, W:

```
Process Covariance (V):
    0.2591    0.0010
    0.0010    0.0625

Sensor Covariance (W):
    1.8817    0.0632
    0.0632    2.1384
```

How V, W were computed:

Process Noise (V)

- We are provided with the linearized approximation in the system:
  - q[k + 1] ≈ F(q[k], u[k])q[k] + G(q[k])u[k] + Γ(q[k])v[k]
- We have access to all terms but the process noise v[k]
- Rearrange the linearized approximation equation such that it yields v[k]
  - v[k] = pinv(Γ(q[k])) * (q[k+1] - F(q[k], u[k])q[k] - G(q[k])u[k])
- Find the covariance of this noise with cov()

Sensor Noise (W):

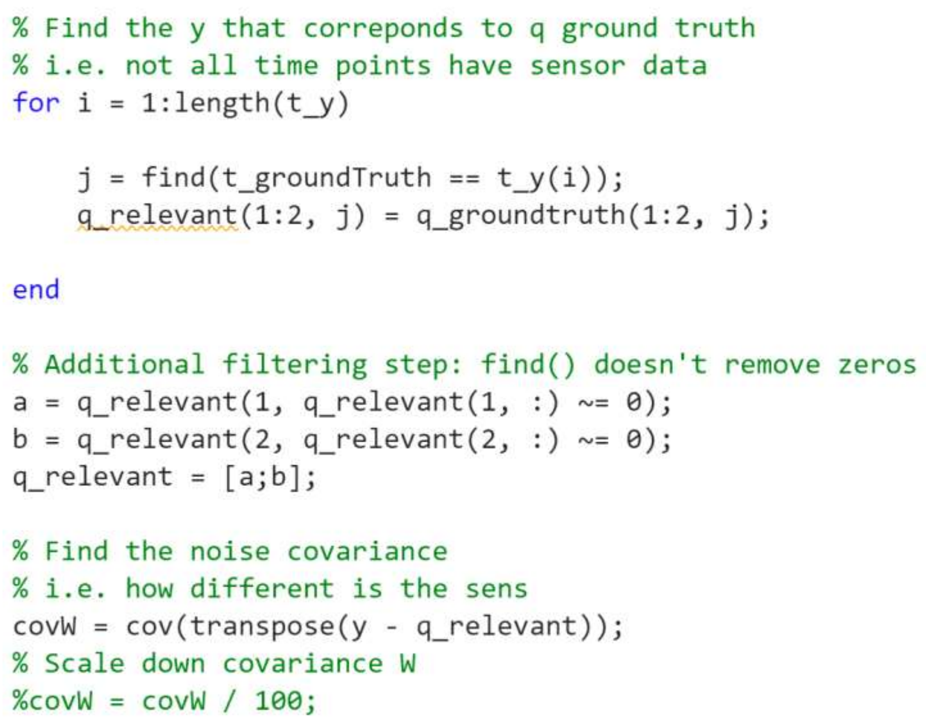
- We are given the ground truth (which is assumed to be perfect) and sensor readings
- The difference between the two will define the sensor noise
- Finding the covariance of the sensor noise will define W (use MATLAB cov() function)

Question 1.b

Tuesday, November 15, 2022 3:44 PM

Sensor Covariance (W):

Code:



```
% Find the y that correponds to q ground truth
% i.e. not all time points have sensor data
for i = 1:length(t_y)

    j = find(t_groundTruth == t_y(i));
    q_relevant(1:2, j) = q_groundtruth(1:2, j);

end

% Additional filtering step: find() doesn't remove zeros
a = q_relevant(1, q_relevant(1, :) ~= 0);
b = q_relevant(2, q_relevant(2, :) ~= 0);
q_relevant = [a;b];

% Find the noise covariance
% i.e. how different is the sens
covW = cov(transpose(y - q_relevant));
% Scale down covariance W
%covW = covW / 100;
```

Explanation:

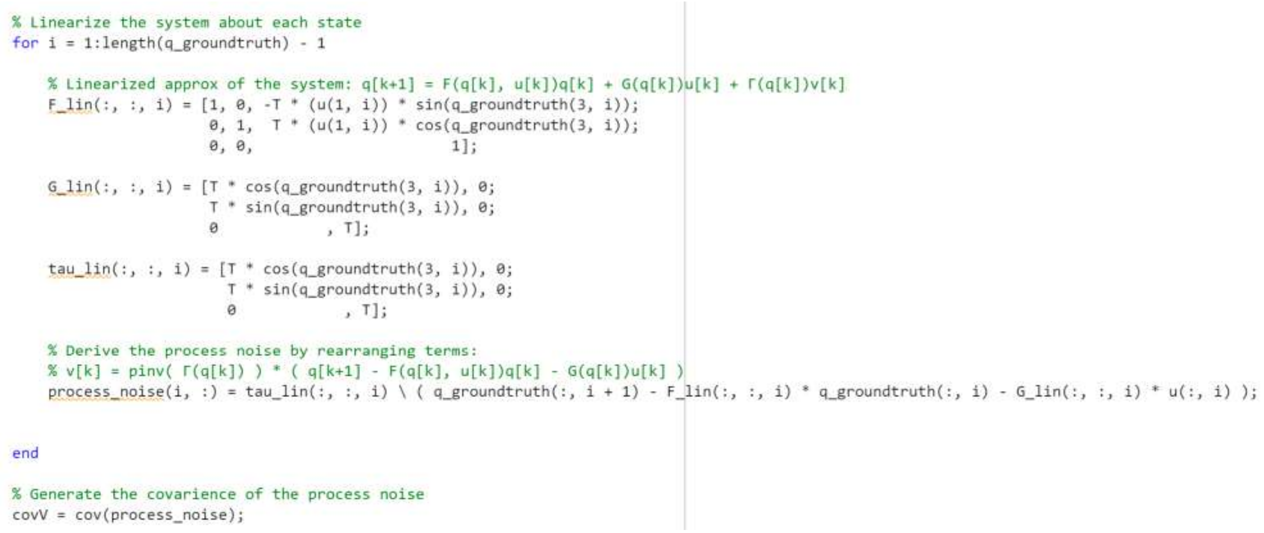
- For all sensor times, find out if that sensor time exists within the ground truth times
- If it does, designate it as a relevant state
- Remove the zeros from q_relevant: if t_y(i) doesn't exist in the t_groundtruth a zero is returned
- Now we have a list of all sensor readings that correspond to the proper state readings
- Find their difference, this is the sensor noise
- Find the covariance of the sensor noise

Question 1.b

Monday, November 14, 2022 12:46 AM

Process Covariance (V)

Code:



```
% Linearize the system about each state
for i = 1:length(q_groundtruth) - 1

    % Linearized approx of the system: q[k+1] = F(q[k], u[k])q[k] + G(q[k])u[k] + Γ(q[k])v[k]
    F_lin(:, :, i) = [1, 0, -T * (u(1, i)) * sin(q_groundtruth(3, i));
                      0, 1,  T * (u(1, i)) * cos(q_groundtruth(3, i));
                      0, 0,                          1];

    G_lin(:, :, i) = [T * cos(q_groundtruth(3, i)), 0;
                      T * sin(q_groundtruth(3, i)), 0;
                      0            , T];

    tau_lin(:, :, i) = [T * cos(q_groundtruth(3, i)), 0;
                        T * sin(q_groundtruth(3, i)), 0;
                        0            , T];

    % Derive the process noise by rearranging terms:
    % v[k] = pinv( Γ(q[k]) ) * ( q[k+1] - F(q[k], u[k])q[k] - G(q[k])u[k] )
    process_noise(i, :) = tau_lin(:, :, i) \ ( q_groundtruth(:, i + 1) - F_lin(:, :, i) * q_groundtruth(:, i) - G_lin(:, :, i) * u(:, i) );


end

% Generate the covarience of the process noise
covV = cov(process_noise);
```

Explanation:

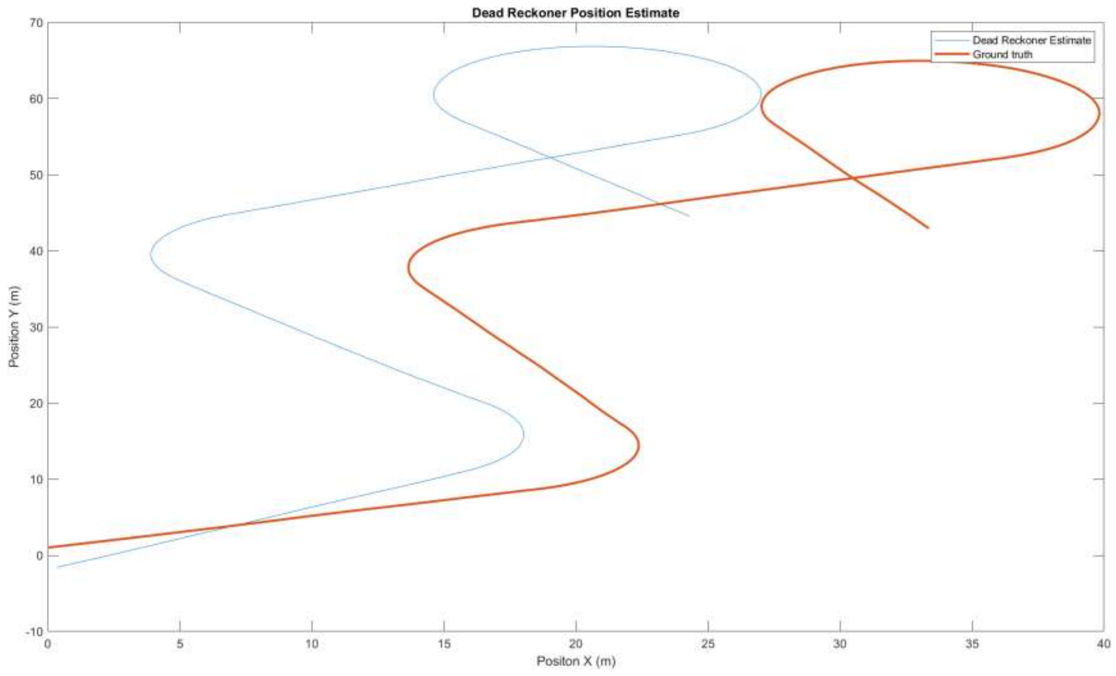
- Find linearization at each state as derived in question 1.a
- Rearrange linearization equation to find process noise at each state
- Using the process noise, find its covariance

Question 1.c.i

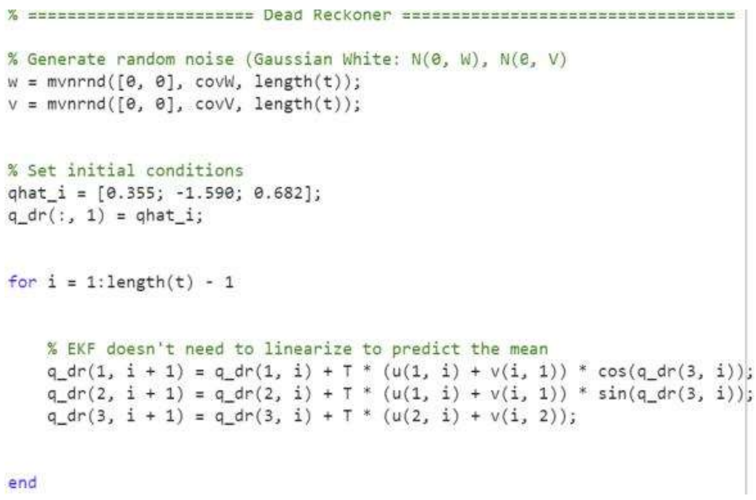
Monday, November 14, 2022   12:47 AM

i. (5 points) Write a dead reckoner, i.e., make an extended Kalman filter that ignores the GPS signal and only does prediction steps. Plot y_r vs. x_r for your dead reckoner on the same plot with y_r vs. x_r for the ground truth. And submit your code.

Plot:



Dead Reckoner Code (Predict Only):



```
% ======================= Dead Reckoner ==================================

% Generate random noise (Gaussian White: N(0, W), N(0, V)
w = mvnrnd([0, 0], covW, length(t));
v = mvnrnd([0, 0], covV, length(t));


% Set initial conditions
qhat_i = [0.355; -1.590; 0.682];
q_dr(:, 1) = qhat_i;


for i = 1:length(t) - 1


    % EKF doesn't need to linearize to predict the mean
    q_dr(1, i + 1) = q_dr(1, i) + T * (u(1, i) + v(i, 1)) * cos(q_dr(3, i));
    q_dr(2, i + 1) = q_dr(2, i) + T * (u(1, i) + v(i, 1)) * sin(q_dr(3, i));
    q_dr(3, i + 1) = q_dr(3, i) + T * (u(2, i) + v(i, 2));


end
```

Question 1.c.ii

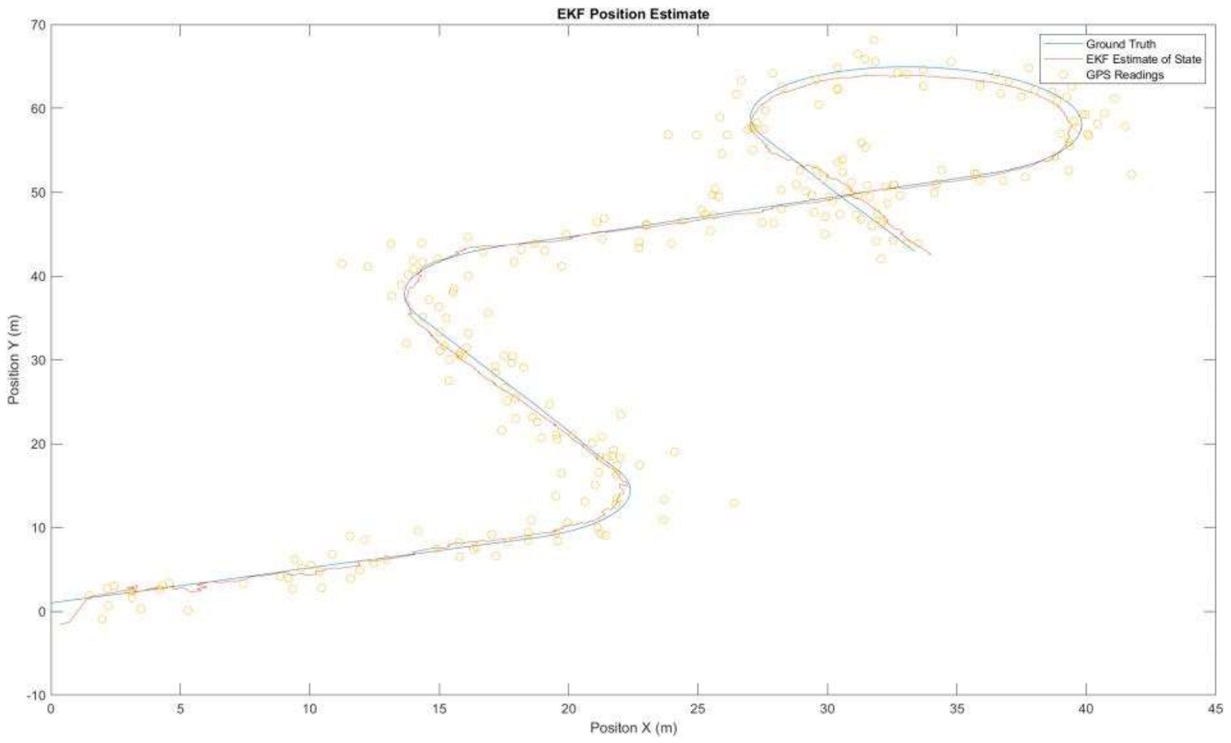
Monday, November 14, 2022 12:53 AM

ii. (25 points) Write a full extended Kalman filter. Submit your code, and make a plot that contains the following:

- a trajectory plot of y_r vs. x_r for the ground truth.
- a scatter plot or y_r vs. x_r from the GPS measurements.

- a trajectory plot of y_r vs. x_r for your EKF.

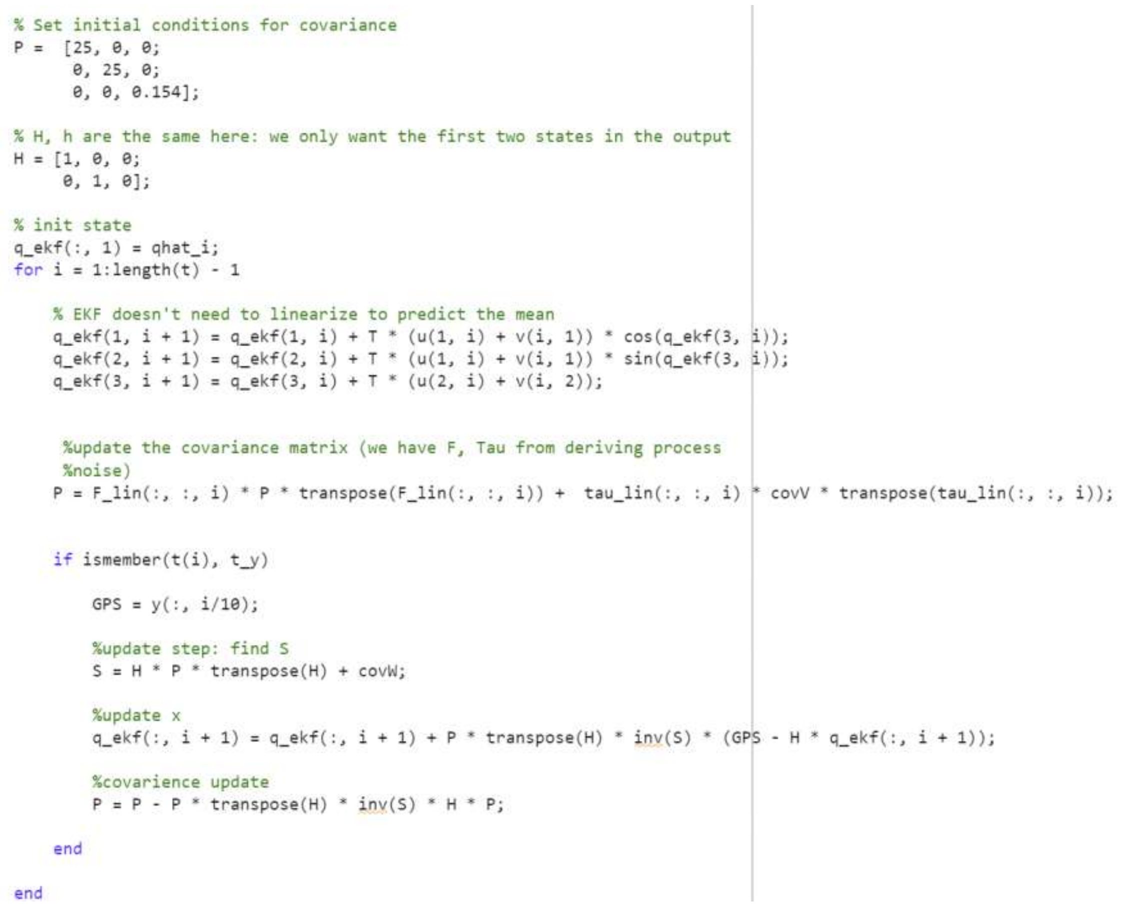
Output:



Question 1.c.ii

Monday, November 14, 2022 12:54 AM

Matlab Code:



```
% Set initial conditions for covariance
P =  [25, 0, 0;
      0, 25, 0;
      0, 0, 0.154];

% H, h are the same here: we only want the first two states in the output
H = [1, 0, 0;
     0, 1, 0];

% init state
q_ekf(:, 1) = qhat_i;
for i = 1:length(t) - 1

    % EKF doesn't need to linearize to predict the mean
    q_ekf(1, i + 1) = q_ekf(1, i) + T * (u(1, i) + v(i, 1)) * cos(q_ekf(3, i));
    q_ekf(2, i + 1) = q_ekf(2, i) + T * (u(1, i) + v(i, 1)) * sin(q_ekf(3, i));
    q_ekf(3, i + 1) = q_ekf(3, i) + T * (u(2, i) + v(i, 2));


     %update the covariance matrix (we have F, Tau from deriving process
     %noise)
    P = F_lin(:, :, i) * P * transpose(F_lin(:, :, i)) +  tau_lin(:, :, i) * covV * transpose(tau_lin(:, :, i));


    if ismember(t(i), t_y)

        GPS = y(:, i/10);

        %update step: find S
        S = H * P * transpose(H) + covW;

        %update x
        q_ekf(:, i + 1) = q_ekf(:, i + 1) + P * transpose(H) * inv(S) * (GPS - H * q_ekf(:, i + 1));

        %covarience update
        P = P - P * transpose(H) * inv(S) * H * P;

    end

end
```

Matlab Code Explanation:

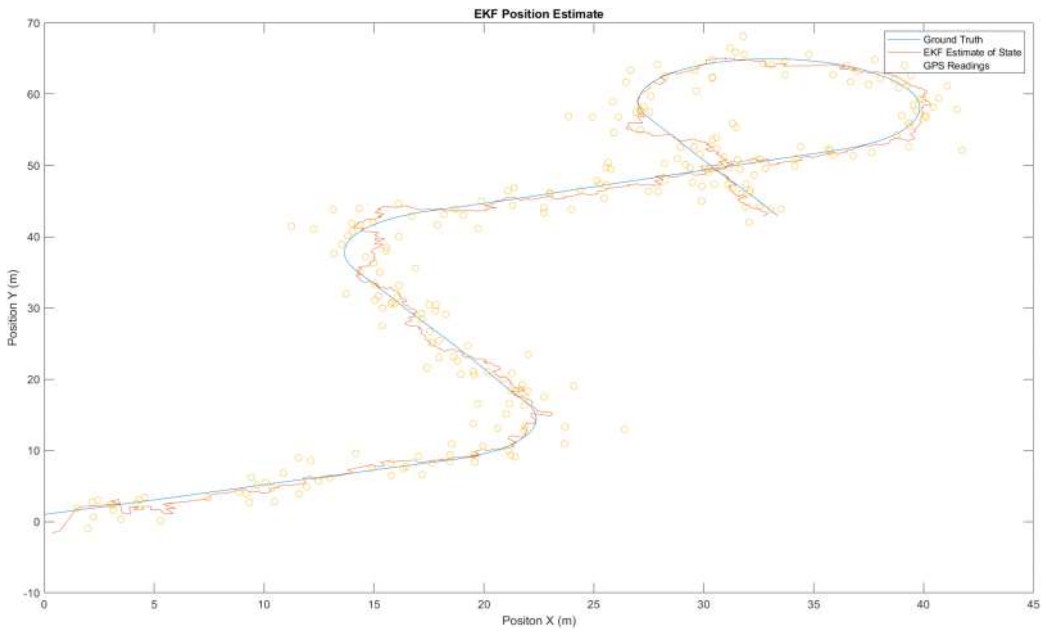
- Initialize the covariance and the state
- Predict step:
  - the EKF doesn't require linearization to predict the mean
  - However, we do need to linearize to predict the covariance
    - F_lin, tau_lin were computed to derive process noise
    - P[k + 1|k] = F P[k|k]F T + V [k]
- Update step (note H, h are the same in this scenario)
  - We will only update the state and covariance when the sensor readings come in, and use the dead reckoner (prediction step only) for all times in between
    - S = HP[k + 1|k]HT + W[k]
    - xˆ[k + 1|k + 1] = ˆx[k + 1|k] + P[k + 1|k]HT S −1 (y[k + 1] − h (ˆx[k + 1|k]))
    - P[k + 1|k + 1] = P[k + 1|k] − P[k + 1|k]HT S −1HP[k + 1|k].

Question 1.c.iii

Monday, November 14, 2022 12:55 AM

iii. (5 points) Rerun your EKF, only this time, scale W down by a factor of 100. Generate all of the same plots. Explain why the difference between this plot and the prior one makes sense.

Output:



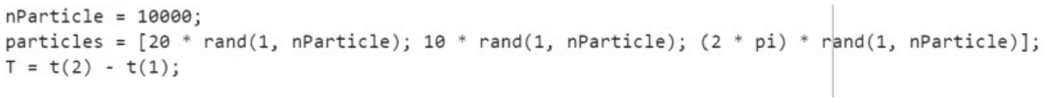
Explanation:

Scaling down the covariance W will tell us that we expect there to be very little noise in the sensor readings. The EKF will trust the sensor measurements more, resulting in the more jerky estimate as the EKF looks to take into account the sensor reading more in its estimate.

Question 2

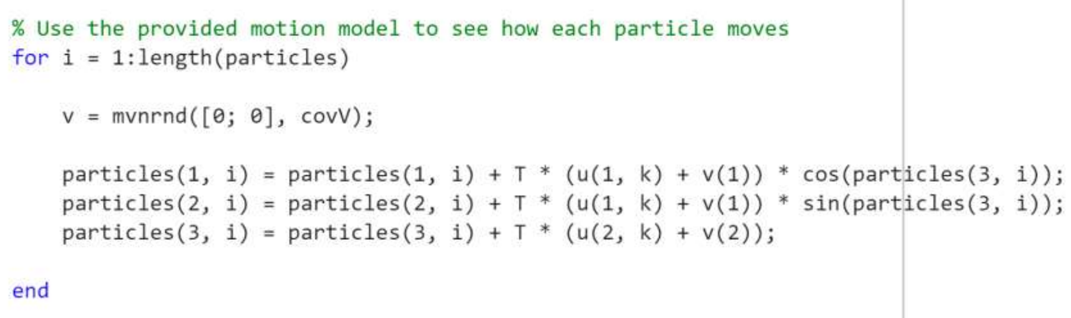
Monday, November 14, 2022 12:58 AM

Initialization:



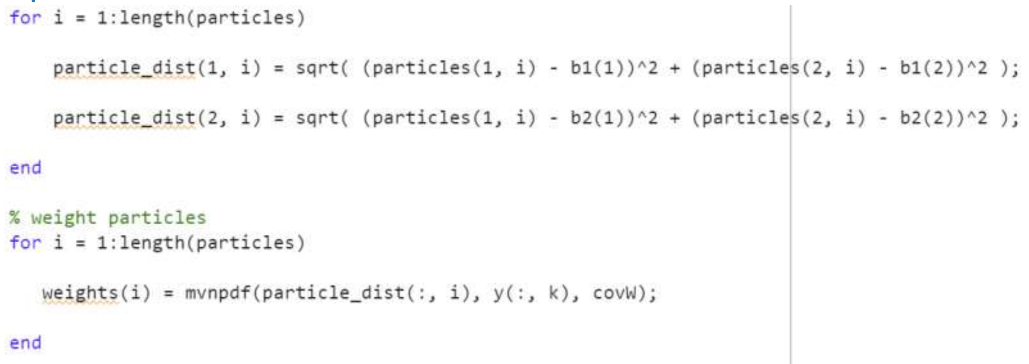
```
nParticle = 10000;
particles = [20 * rand(1, nParticle); 10 * rand(1, nParticle); (2 * pi) * rand(1, nParticle)];
T = t(2) - t(1);
```

Prediction:



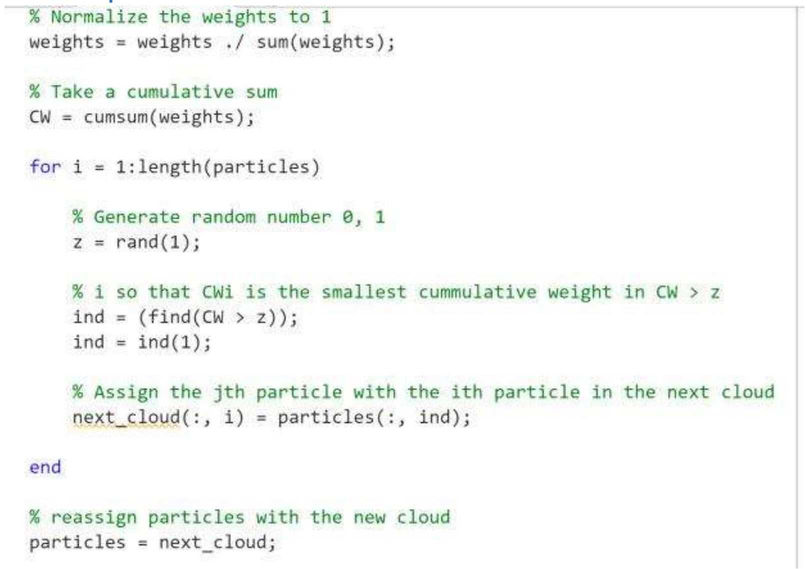
```
% Use the provided motion model to see how each particle moves
for i = 1:length(particles)

    v = mvnrnd([0; 0], covV);

    particles(1, i) = particles(1, i) + T * (u(1, k) + v(1)) * cos(particles(3, i));
    particles(2, i) = particles(2, i) + T * (u(1, k) + v(1)) * sin(particles(3, i));
    particles(3, i) = particles(3, i) + T * (u(2, k) + v(2));

end
```

Update:



```
for i = 1:length(particles)

    particle_dist(1, i) = sqrt( (particles(1, i) - b1(1))^2 + (particles(2, i) - b1(2))^2 );

    particle_dist(2, i) = sqrt( (particles(1, i) - b2(1))^2 + (particles(2, i) - b2(2))^2 );

end

% weight particles
for i = 1:length(particles)

   weights(i) = mvnpdf(particle_dist(:, i), y(:, k), covW);

end
```

Resample:



```
% Normalize the weights to 1
weights = weights ./ sum(weights);

% Take a cumulative sum
CW = cumsum(weights);

for i = 1:length(particles)

    % Generate random number 0, 1
    z = rand(1);

    % i so that CWi is the smallest cummulative weight in CW > z
    ind = (find(CW > z));
    ind = ind(1);

    % Assign the jth particle with the ith particle in the next cloud
    next_cloud(:, i) = particles(:, ind);

end

% reassign particles with the new cloud
particles = next_cloud;
```

Question 2

Tuesday, November 15, 2022 4:27 PM

Code Explanation

Initialization Step

- In the initialization step, the particles were generated to uniformly distribute the state space of x positions, y positions, and orientations theta
  - In the general form it calls for the weights to be initialized to one to reflect that all particles are equally likely, but in my code structure all weights are reassigned at each iteration, which would make a weight initialization useless. It should be noted that each state still has equal likelihood at the first time step
  - Also the time step for each iteration was taken as the difference between two subsequent time vector values
  - Also within the initialization step are the covariance tuning for process and sensor variable, which will be discussed in a few pages

Prediction Step

- First a random noise was calculated with the tuned process noise covariance
- The prediction step leveraged the motion model to predict the next value of each particle, with the motion model being the nonlinear model provided in problem one.

Update Weights

- Weight the particles with the according to the pdf with expectation being the current sensor reading and the covariance being the sensor covariance W.

Resample

- Normalize the weights to one
- Take a cumulative sum of the weights
- For all i particles resample:
  - Make a random number generator
  - Find all the places where the cumulative weight vector is greater than z
  - Take the first index from the above result
  - Use that index to reassign the ith particle of the next cloud to be the indexed position of the current cloud

Note:

For the simulation the red dot being plotted is the "best estimate" at the current iteration, taken as the highest weighted particle of the current iteration.

The blue dot is the average of all the particle states, this gives a fair estimate of the state once the cloud converges to a general location. ***The blue dot was added right before submission, so it does not appear on figures but simply the final simulation. Note that no other parameters were changed to add this aspect to the visualization.***

Question 2

Monday, November 14, 2022 1:05 AM

Finding Process Covariance:

To gain some intuition of how to tune the process I started with a reasonable number of particles (500) and started with the identity matrix for both V, W. For simplicity sake, I took the off diagonal terms to be zero, signifying that they are independent of each other.

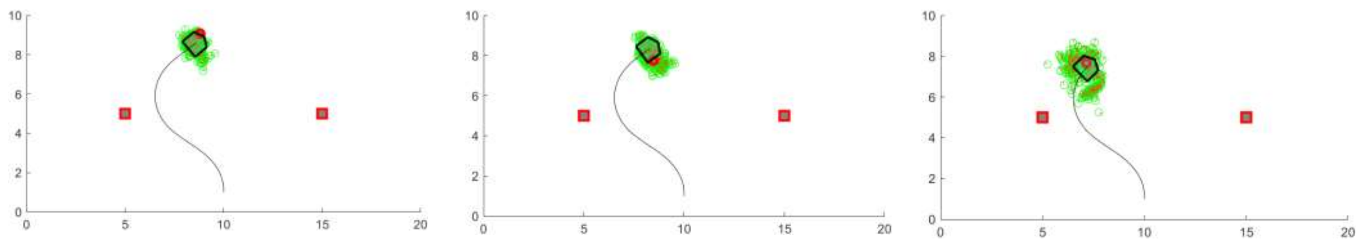


Fig. *Particle Filter output for (similar frames) for covariance of [0.5, 0; 0, 0.5] (left), [1, 0; 0, 1] (center), and [2, 0; 0, 2] (right).*

From this small testing cycle I was able to observe that increasing the process noise caused the particles to explore the environment to a higher degree. For this particular application it seemed that an upper bound for tuning should be two for the diagonal of the process covariance.

Finding Sensor Covariance

The same process for finding bounds on the sensor covariance was used here, giving similar results as tuning the covariance for the sensor noise. Again, giving me the intuition not to tune my covariance above 2 for the diagonals of the matrix.

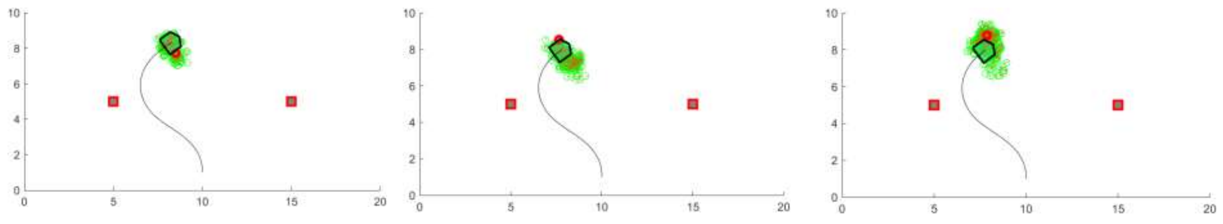


Fig. *Particle Filter output (similar frames) for covariance of [0.5, 0; 0, 0.5] (left), [1, 0; 0, 1] (center), and [2, 0; 0, 2] (right).*

After doing these tunings, I also went on to investigate what happens when these matrices are changed to really small and really big diagonals. With two low covariances diagonals, the particles inadvertently trust incorrect particles and incorrectly assuming there to be no variance in the motion model or sensor measurements (see next page). With large diagonals, the particles fail to converge to a singular point because they don't trust either process or sensor information.

Question 2

Tuesday, November 15, 2022 4:20 PM

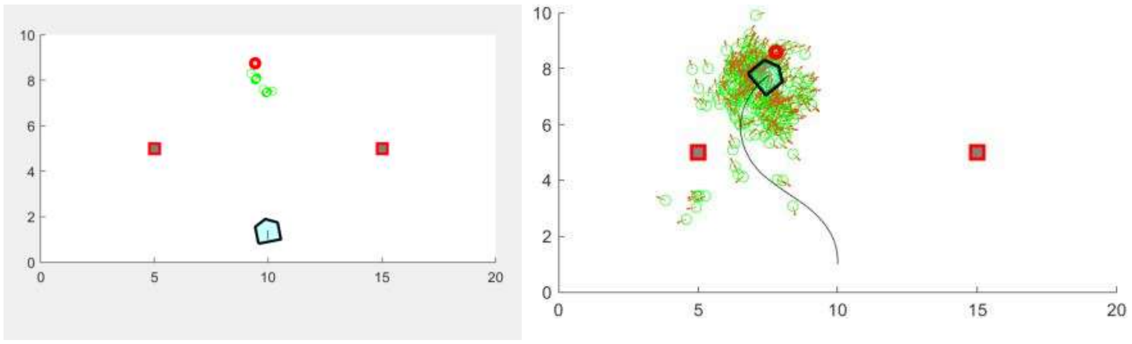


Fig. *Particle Filter output for both covariances with small diagonals of 0.1 (left) and large diagonals of 5 (right).*

After gaining this intuition on tuning the covariances, I spent some time tuning and landed on final process noise and sensor noise of [1.1, 0; 0, 1.25] and [0.5, 0; 0, 0.5]. This gave a satisfactory cloud of particles for my state estimate, but I was skeptical that 500 particles were required to give a reasonable estimate; being able to reduce the number particles would increase computational speed by reducing resources.

How many particles to use?

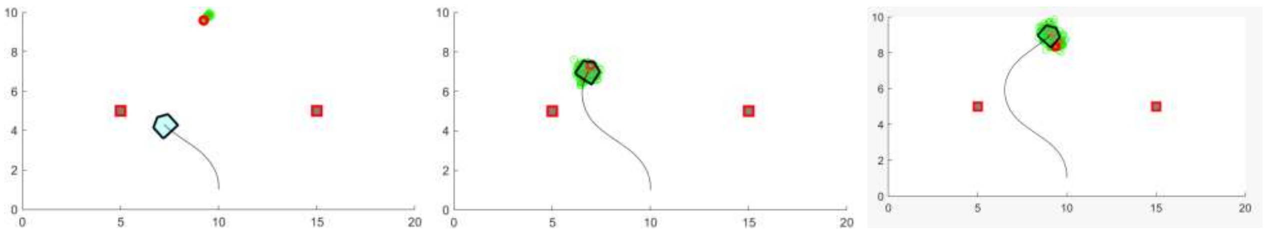


Fig. *Particle Filter output for 100 particles (left), 300 (center), and 500 (right).*

With the idea of mind to prune down the number of particles, I continually decreased the number of particles until I saw a decrease in performance, and saw such a decrease below roughly 300 particles. This is the final value of the particles used for the simulation.