CRANFIELD UNIVERSITY

ASSIGNMENT – VII

AUTONOMY IN ROBOTIC SYSTEMS

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING

Mr. Balasekhar Chandrasekaran Kannan

S359368

M.Sc. Robotics

Academic Year : 2021 – 2022

Faculty : Dr. Leonard Felicetti

April 2022

ABSTRACT

Localisation and navigation are important topics in mobile robot research since they determine the efficiency of the process outcomes(Ehab I. Al Khatib, 2015). The processes can be optimised by making use of all the sensors available in the robot. This study is divided into two parts. First part deals with the localisation process of the bicycle robot. This involves sensor fusion of odometry and Lidar measurements using Extended Kalman filter(EKF). The algorithm is applied over six landmarks in order to determine the robot's position. Secondly, we are going to look at path planning and following where Dubins path algorithm is used to geometrically calculate the path which satisfies all the heading angle constraints at the desired points.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. PART-1

2. DESCRIPTION

The robot used for the experimentation is a bicycle robot fitted with odometry and Lidar sensor. For simplicity of calculations, we have assumed that lidar is located at the geometric centre of the robot. The dynamic model of the robot has three state variables namely x-position, y-position and orientation. It relies on two control inputs- linear and angular velocity.

The state vector gives the global position and orientation of the robot.

$$X = [x \quad y \quad \theta]^T$$

The control inputs are given by,

$$U = \begin{bmatrix} v \\ \omega \end{bmatrix}$$

Where v is the velocity and ω angular velocity.

The state equation of the robot with respect to its global position for its increment time step is given by,

$$X_K = X_{K-1} + dt * \begin{bmatrix} cos\theta_{k-1} & 0 \\ sin\theta_{k-1} & 0 \\ 0 & 1 \end{bmatrix} * \left(\begin{bmatrix} v_{K-1} \\ \omega_{K-1} \end{bmatrix} + w_{K-1} \right)$$

Where,

X_K is the predicted state vector

X_{K-1} it the calculated state vector

dt is the incremented time step

v_{K-1} is the control velocity input

ω_{K-1} is the control angular velocity input

w_{K-1} is the error with zero mean and covariance Q

3. PROBLEM STATEMENT

The internal sensors in the robot such as odometry and IMU's are always prone to error despite careful calibration of the sensors. This can be due to slippage of wheels. Irregularities present in the floor surface such as cracks, bumps etc. these features induce unpredictable irregularities when interacting with the wheels of the robot which cannot be eliminated.

The error caused by these interactions are at times assumed to be linear for ease of calculations which is never the case. If the noise produced by these factors are correlated, then multiple sensors can be attached to remove them. The effect of noise reduction is inversely proportional to the square root of number of sensors. The accuracy of our model will only improve if the number of attached sensors is high, but will some point hit a saturation level beyond which error rectification is impossible. And it works only for correlated errors.

In the real world, the error caused will be non-linear and in most of the cases uncorrelated. Instead of increasing the number of sensors, attaching a different sensor which measures a different quantity would be beneficial if successful sensor fusion can be achieved.

3.1. OBJECTIVES

- To develop a sensor fusion algorithm using EKF to fuse odometry and Lidar measurements.
- Formulate the predication step using motion model to calculate the state value using covariance estimate and control input for the given timestep.
- Implement the measurement step using the state values as inputs.
- Calculate the correction step using measurement values and Lidar measurements to correct the state variables and pose covariance estimates.

4. METHODOLOGY

4.1. PREDICTION STEP

The initial step of the Extended Kalman filter algorithm is prediction of next state vector using previous state and control inputs.

$$X_K = f(X_{K-1}, U_{K-1}, w_{K-1})$$

This can be converted from vector form to matrix in the following way,

$$X_K = X_{K-1} + V * U_{K-1}$$

The error associated with the current timestep is calculated using error covariance matrix. Initial value of the covariance matrix can be set using trial and error or depending on our confidence on the model. This will not effect our prediction step since this matrix will be updated after correction. The error in the prediction step is a result of contribution by error propagation of previous step obtained by covariance of previous time step and error in the control input defined by matrix V.

$$P_K = G * P_{K-1} * G^T + V * Q * V^T$$

The matrices G and V are state transition matrix and input transition matrix. G can be calculated by taking the Jacobian of dynamic model equations with respect to state variables. It is n*n matrix where n is the number of state variables which are x, y and θ here.

$$G = \begin{bmatrix} 1 & 0 & -v_{K-1} * dt * sin\theta_{k-1} \\ 0 & 1 & v_{K-1} * dt * cos\theta_{k-1} \\ 0 & 0 & 1 \end{bmatrix}$$

V is the matrix responsible for translating input variables to state variables and is a n*m matrix where m is the number of control inputs.

$$V = \begin{bmatrix} dt * cos\theta_{k-1} & 0 \\ dt * sin\theta_{k-1} & 0 \\ 0 & dt \end{bmatrix}$$

The initial covariance matrix is given as,

$$P_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix}$$

And the input covariance matrix Q is given as

$$Q = \begin{bmatrix} 0.01 & 0 \\ 0 & 0.25 \end{bmatrix}$$

Where, the variance of input velocity is 0.01 and variance of angular input is 0.25.

Using V matrix, the state vector for next timestep can be predicted and the error covariance is updated for the current timestep which will be used to calculate Kalman gain in correction step.

4.2. MEASUREMENT STEP

The measurements obtained from the Lidar is range and bearing angle and the odometry values has to be converted to these values for sensor fusion. A measurement model is defined to convert state vector to measurement vector which gives the pose of the bicycle bot with respect to the landmarks.

$$Z_K^l = [r \quad \varphi]^T$$

Where Z_K^l is the output from measurement model and l is the landmark and K is the timestep.

The description of measurement model is,

$$Z_K^l = H * X_K$$

$$Z_K^l = \begin{bmatrix} \sqrt{((x_l - x_K)^2 + (y_l - y_K)^2)} \\ atan2(y_l - y_K, x_l - x_K) \end{bmatrix}$$

Here H is the measurement matrix which gives the relationship between the state vector and output variables. It has a dimension of m*n which is 2*3 in this case. The current state values are the input and the measurement values are obtained for all the landmarks present in the map.

$$H = \begin{bmatrix} -(x_l - x_K)/\sqrt{((x_l - x_K)^2 + (y_l - y_K)^2)} & (y_l - y_K)/\sqrt{((x_l - x_K)^2 + (y_l - y_K)^2)} & 0 \\ (y_l - y_K)/(x_l - x_K)^2 + (y_l - y_K)^2) & -(x_l - x_K)/(x_l - x_K)^2 + (y_l - y_K)^2 & -1 \end{bmatrix}$$

4.3. CORRECTION STEP

This step along with the measurement step will be calculated for all the landmarks in the map. This enables us to get a more accurate localisation rather than using a single landmark.

The first process is to calculate the innovation of the range and bearing values which is the difference between measurement model value and Lidar value. This is also called as a residual value.

$$dZ = Z - Z_K^l$$

The innovation covariance is calculated using the following step,

$$S = H * P_K * H^T + R_K^l$$

The R_K^l is the sensor measurement matrix and is the parameters are usually given by the sensor manufacturers. The variance of sensor measurements can also be obtained by repetitive measurement and calibration. In our case of 6 landmarks, the first 3 have a different range variance from the last 3 whereas the heading angle variance remains constant at 0.25.

$$R_K^{1\_3} = \begin{bmatrix} 0.01 & 0 \\ 0 & 0.25 \end{bmatrix}$$

$$R_K^{4\_6} = \begin{bmatrix} 0.09 & 0 \\ 0 & 0.25 \end{bmatrix}$$

Information matrix is the inverse of innovation covariance and this step is the most computationally expensive step in EKF process.

$$I = S^{-1}$$

Kalman gain determines which measurement is valued i.e. measurement model value or Lidar measurement.

$$K_G = P_K * H^T * I$$

The predicted state value and error covariance is updated using the obtained Kalman gain.

$$X = X_K + K_G * dZ$$

$$P = P_K - K_G * H * P_K$$

This calculation of Kalman gain and updating process runs in an iterative loop for all the landmarks present for a single timestep and final value after all landmarks is the correct state value and error covariance matrix.

5. RESULTS AND DISCUSSION

The Extended Kalman filter algorithm was applied for the bicycle bot whose localisation was dependent on six landmarks present in the surrounding and the encoder and Lidar measurements. The figure below shows the x and y position of the robot at every time step. It also shows the location of the landmarks as small circles. Initial estimation error of position can be seen in the jump at first few steps which was corrected instantly by applying the fusion algorithm.

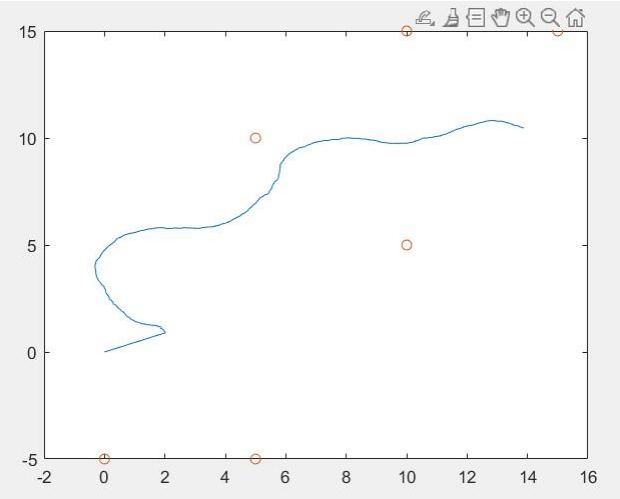

Figure 1 : Position of bicycle bot and landmark locations

6. PART-2

7. DESCRIPTION

One of the challenges of an Unmanned Ground Vehicle(UGV) is to find the past to the desired point on its own and follow it. It is necessary for the UGV to reach the points at a certain orientation for the task to be successful. The given UGV has a velocity of 5 m/s, maximum lateral acceleration of 5 m/s^2 and a turn radius of 5m at every waypoint.

The motion equations of UGV in 2D are given by

$$\dot{x} = v_a cos\alpha$$

$$\dot{y} = v_a sin\alpha$$

$$\dot{\alpha} = u/v_a$$

	Position (m)	Desired Heading Angle (°)
Initial Point	$(0 \quad 10)^T$	0
Waypoint 1	$(60 \quad 60)^T$	45
Waypoint 2	$(80 \quad 120)^T$	30
Waypoint 3	$(150 \quad 70)^T$	-90
Waypoint 4	$(100 \quad 30)^T$	-120
Waypoint 5	$(50 \quad 0)^T$	-180

Figure 2 : Location and Heading angle of waypoints for Path generation

8. PROBLEM STATEMENT

An UGV travelling between two points has to satisfy many constraints in order to start from the initial point at a certain orientation and reach the final point at specified orientation. These constraints would help us get the paths available to reach a location but says nothing about the path following.

8.1. OBJECTIVES

- Designing a path using Dubins Path algorithm for the specified waypoints given in figure 2.
- Generate a path following algorithm for the above generated path

9. METHODOLOGY

9.1. PATH PLANNING

Dubins path algorithm was used geometrically to calculate the exit and entry angles and locations between waypoints. A Mat-Lab code named Dubins_Path_Planning-geometry.m calls another file Dubins_Path.m was written to calculate the Dubins circles and their centre point and other constraints. The file is run for every waypoint and it is added in the appendix for reference.

The initial step is to calculate the centres of the Dubin circles using the position and orientation of initial and final waypoints. Our case doesn't have a secondary circle since the radius across all waypoints is same. Distance between two circle centres is calculated and exit entry points and angles are also calculated. This step process is continued between all the consecutive waypoints for complete path generation.

9.2. PATH FOLLOWING

The next step after planning a path is to make the UGV follow it. Carrot chasing algorithm is implemented to make the UGV follow the path where a vector field is created which is updated as the time progresses. The UGV follows this vector field to follow the path successfully.

Three Mat-Lab files are written in order to make the UGV follow the path. Carrot_Chasing_algorithm.m is the main program and it calls CCA_Dubin_Circle.m and CCA_Straight_Path.m to complete the task.

The main program first calculates the path using Dubins_Path.m and calls CCA_Dubin_Circle.m for guiding the UGV through the first Dubin circle between waypoints.

The initial step is to enter all the parameters and calculate the angle between the waypoints. If the absolute value of angle difference between waypoints and from current position to final waypoint is within the specified parameter, then the new position of the UGV is updated. Heading angle for the travel is calculated and the acceleration is kept under check. Dynamic model and state updates are done for the current timestep. This process is iterated until the exit point from the Dubin circle is reached.

The Carrot_Chasing_algorithm.m then calls CCA_Straight_Path.m to follow path between the exit point of the first Dubin circle and entry point of the second circle. The code runs till the line is drawn till the entry point. The distance between initial waypoint and current position,

orientation between them, orientation between waypoints and the distance between waypoints along the projection of the path are calculated. Using these values, the carrot path position and heading angle is updated at every timestep. The dynamic model and state update is done by controlling the acceleration and heading angle. Here the code is made flexible to draw a straight line between two waypoints regardless of if it is starting from a Dubin circle or ending in one.

The control goes back to the main program and CCA_Dubin_Circle.m is called once again to draw the second Dubin circle. This process is continued until all the paths between waypoints are followed.

10. RESULTS AND DISCUSSION

The below figure shows the path planning done geometrically by Dubin circle for all the waypoints given. Table 1 also gives the required parameters for understanding the path generation between 2 points.

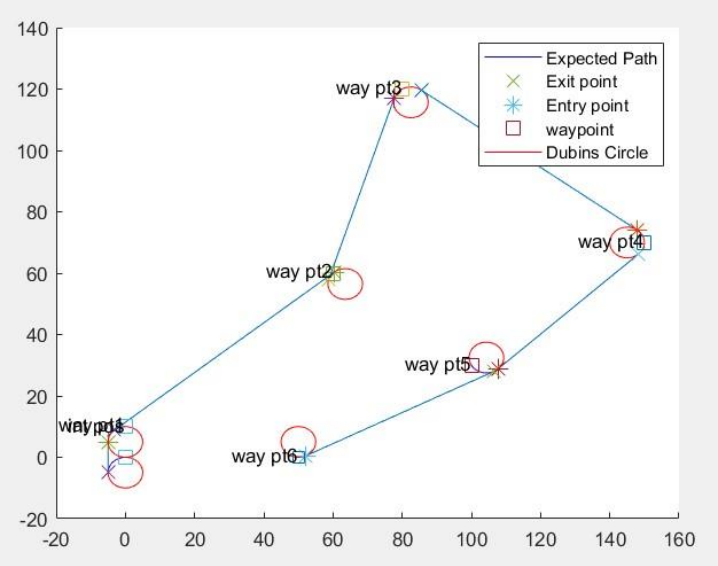

Figure 3 : Dubin Path Planning

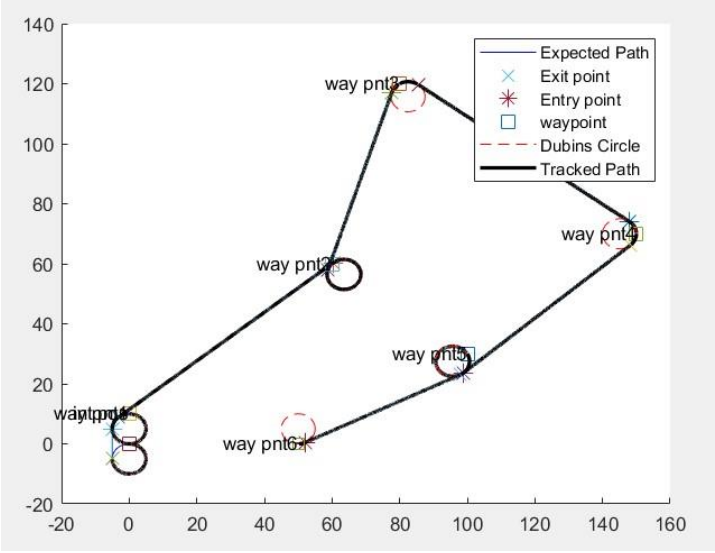

Figure 4 : Carrot Chasing Algorithm - Path Following

Waypoints	Entry Circle Centre	Exit Circle Centre	Entry Angle	Exit Angle	Entry Location	Exit Location
Initial point	(3e-16, 3e-16)	(-5, 5)	3.14	3.14	(-5, -5)	(-5, 5)
Way pt1	(3e-16, 63.5)	(5, 56.4)	2.25	2.25	(-3.14, 8.8)	(60.3, 60.3)
Way pt2	(63.5, 82.5)	(56.4, 115.6)	2.83	2.83	(58.7, 57.9)	(77.7, 117.1)
Way pt3	(82.5, 145)	(115.6, 70)	0.93	0.93	(85.4, 119.7)	(147.9, 74.0)
Way pt4	(145, 104.3)	(70, 32.5)	-0.82	-0.82	(148.3, 66.3)	(107.7, 28.8)
Way pt5	(104.3, 50)	(32.5, 5)	-1.10	-1.10	(106.5, 28.03)	(52.2, 0.53)

Table 1 : Values of all Waypoints

The path following was able to preform successfully and the path taken by the UGV is given in bold line in figure 4. The simulation also shows the path following as the time step increases.

Various parameters such as carrot distance, gain were varied to see their effects for this particular case. The effects of gain weren't dominant enough to make a difference. Whereas the carrot distance affected the accuracy of the path taken by the UGV.

The carrot distance did not have any significant effect in CCA circular path but had huge deviations in straight path motion.

The below image shows the comparison of change in carrot distance from 1e6 to 1e2 through different stages.

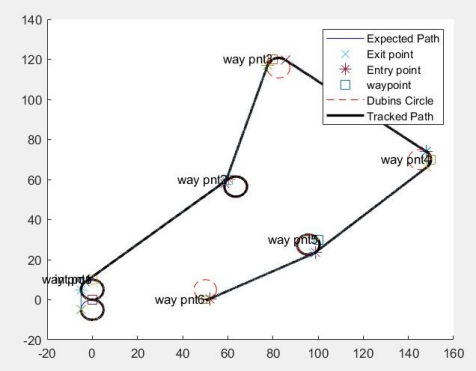

Figure 5 : Carrot Distance - 1e5

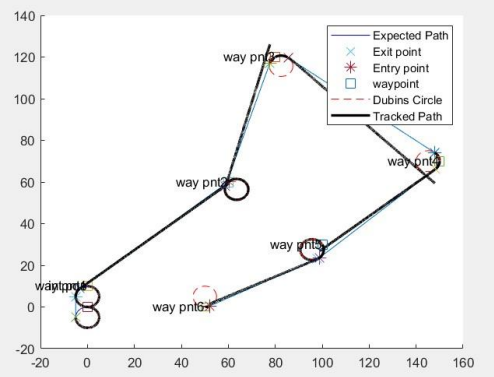

Figure 7 : Carrot Distance - 1e3

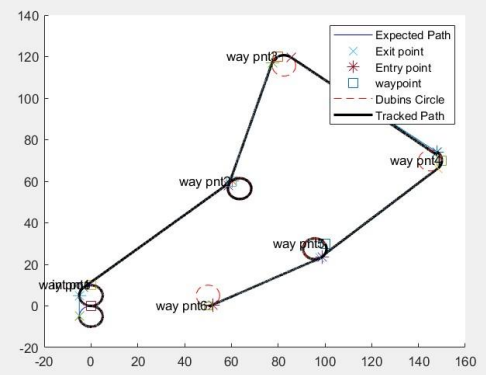

Figure 6 : Carrot Distance - 1e4

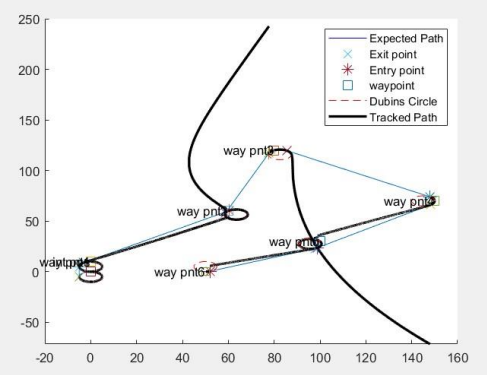

Figure 8 : Carrot Distance - 1e2

There is almost no change in the path followed by the UGV when the carrot distance is 1e5 by we can see a slight change when the distance changes to 1e4 at waypoint 4. Huge deviations were seen when the distance is 1e3 as shown in figure 7. Any reduction of carrot distance below this confuses the UGV completely and the trajectory follow is not even a straight line anymore. This is supported by figure 8.

11. CONCLUSION

In this study we were able to successfully fuse the odometry and Lidar readings of a bicycle bot using Extended Kalman Filter and achieve localisation. This was achieved by comparing the global position of the bot against six landmarks provided.

Successful path generation and following by UGV was done for the specified waypoints and the effects of different parameters was compared using Mat-Lab simulations. Dubins Path was implemented and the path was geometrically calculated for the UGV. Carrot Chasing algorithm was then used to make the UGV follow the path without error.

The code necessary for the simulations are provided in the appendix section for validation and proof of results provided.

12. REFERENCES

Ehab I. Al Khatib, M. A. (2015). Multiple sensor fusion for mobile robot localizationand navigation using the Extended Kalman Filter. Sharjah.

13. APPENDIX

13.1. APPENDIX-1

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                    %
%   EKF_Localisation                                 %
%                                                    %
%                                                    %
%                  Created by Balasekhar CSK         %
%                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Performs sensor fusion of Odometry and Lidar values

%Loading control inputs and Lidar measurements
clear variables
load my_input.mat
load my_measurements.mat


%Initialising the values with given measurements of variance of sensors
t=t';
dt = t(2)-t(1);
v=v';
om=om';

sigma_velocity=0.01;
sigma_angular_velocity=0.25;
sigma_range_13=0.01;
sigma_range_46=0.09;
sigma_bearing=0.25;

%Array for storing pose of robot at every timestep
x = zeros(3,length(t));

%Array for storing error covariance of robot at every timestep
P = zeros(size(x,1),size(x,1),length(t));
%Initialising the first error covariance
P(:,:,1)=diag([1 1 0.1]);

%Initialising the sensor measurement matrix and control input matrix
Q=diag([sigma_velocity sigma_angular_velocity]);
R_13=diag([sigma_range_13 sigma_bearing]);
R_46=diag([sigma_range_46 sigma_bearing]);

ChiStat = zeros(1,length(t));

%Looping through all the timesteps
for k = 2:length(t)

    %Jacobian of state equation with respect to state veriables
    G=[1  0  -dt*v(k-1)*sin(x(3,k-1));
       0  1  dt*v(k-1)*cos(x(3,k-1));
       0  0  1];

    %Jacobian of state equation with respect to control input
    V=[dt*cos(x(3,k-1))  0;
       dt*sin(x(3,k-1))  0;
```

```
    0                dt];

    %Control inputs
    Input=[v(k-1);
           om(k-1)];

    %Estimation of state variable from previous time step
    %Prediction step
     xa = x(:,k-1) + V*Input;

     %Predicted error covariance

     Pa = G*P(:,:,k-1)*G'+V*Q*V';

     %Performing measurement and correction step for all the six landmarks
     for i=1:length(l)

         if i<4

         [Pa,xa,ChiStat(1,k)]=measurement(xa,Pa,l(i,:),b(k-1,i),r(k-1,i),R_13);

         else

          [Pa,xa,ChiStat(1,k)]=measurement(xa,Pa,l(i,:),b(k-1,i),r(k-1,i),R_46);
         end
         P(:,:,k)=Pa;
         x(:,k)=xa;
     end
end

%figure for Plotting the values

plot(x(1,:),x(2,:));
hold on;
scatter(l(:,1),l(:,2));
```

13.2. APPENDIX-2

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                    %
%   Measurement                                      %
%                                                    %
%                                                    %
%                  Created by Balasekhar CSK         %
%                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function   [P,x,ChiStat]=measurement(x_a,P_a,land,b,range,R)

z=[range, b]';
d=(land(1)-x_a(1,1))^2+(land(2)-x_a(2,1))^2;

%Predicted measurements using a priori states
za=[d^0.5;
    wrapToPi(atan2((land(2)-x_a(2,1)),(land(1)-x_a(1,1))))-x_a(3,1)];
% Jacobian matrix: linearization of the measurement model
    % Dimensions: m x n: m - number of measurements / n - number of states
 H=[-(land(1)-x_a(1,1))/d^0.5 -(land(2)-x_a(2,1))/d^0.5 0;
    (land(2)-x_a(2,1))/d      -(land(1)-x_a(1,1))/d     -1];

% Innovation covariance
S = H*P_a*H'+R;

%Information matrix of innovations
Yzz = S^(-1);

% Calculation of Kalman gain
K = P_a*H'*Yzz;

%Innovations
dz = z-za;

% State update and Error covariance update
x = x_a + K*dz;
P = P_a - (K * H * P_a);
ChiStat = dz'*Yzz*dz;

end
```

13.3. APPENDIX-3

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                %
%   Dubins Path Planning                         %
%                                                %
%                                                %
%                  Created by Balasekhar CSK     %
%                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clc;
clear all; %#ok<CLALL>
close all;

% starting position
int_pos = [0,0,0];

% list of waypoints in the format [x,y,heading angle]
way_pnt = [0,10,0;
        60,60,pi/4;
        80,120,pi/6;
        150,70,(-pi/2);
        100,30,(-4*pi/3);
        50,0,-pi];

radius = 5;

%Parameters that will be calculated while estimating dubins path
%Centre of two Dubin circles, entry and exit angle and position
param(1,:) = ["Xcs", "Ycs", "Xcf", "Ycf", "Tx_x", "Tx_y", "Tn_x",
"Tn_y","phi_ex","phi_en"];

% to estimate for all the waypoints
for i = 1:length(way_pnt)

    %Initialising the first point from where UGV will Travel
    xi = int_pos(1,1); yi = int_pos(1,2); theta_s = int_pos(1,3);

    %Initialising the final point of reach
    xf = way_pnt(i,1); yf = way_pnt(i,2); theta_w = way_pnt(i,3);
    way= way_pnt(i,:);

    % Dubins_Path is user defined function for dubins which returns
    % constraints
    [xcs,xcf,ycs,ycf,Tx,Tn,phi_ex,phi_en] = Dubins_Path(int_pos,way,radius);

    %Plot the path estimated by Dubin algorithm
    [h,h1,h2,h3,h4,thetad] =
plotter(xcs,xcf,ycs,ycf,radius,Tx,Tn,[xi,yi,theta_s],[xf,yf]);

    % update initial point after every loop
    int_pos = way_pnt(i,:);

    % Store all the values
    param(i+1,:) =  [xcs,xcf,ycs,ycf,Tx(1),Tx(2), Tn(1), Tn(2),phi_ex, phi_en];

end
```

```
plot(int_pos(1),int_pos(2),'s','MarkerSize',8,'DisplayName','Point');
xx = 0;yy =10.5;
for j = 1:length(way_pnt)
    xx = [xx,way_pnt(j,1)]; %#ok<AGROW>
    yy = [yy,(way_pnt(j,2)+1)]; %#ok<AGROW>
end

%to label all the waypoints UGV is going to travel through
labels = {"int pos","way pt1","way pt2","way pt3","way pt4","way pt5","way pt6"};
%#ok<CLARRSTR>
text(xx,yy,labels,'VerticalAlignment','HorizontalAlignment','right');
legend([h h1 h2 h3 h4], {"Expected Path", "Exit point","Entry point",
"waypoint","Dubins Circle",  "Path"}); %#ok<CLARRSTR>

hold off
disp(param)



%Funtion  to plot the path estimated by Dubin algorithm
function [h,h1,h2,h3,h4,thetad] = plotter(xcs,xcf,ycs,ycf,radius,
Tx,Tn,int_pos,fin_pos)

    [h,h4,thetad] = Dubin_circle(xcs,ycs,radius,int_pos,Tx);
    hold on


    line([Tx(1),Tn(1)],[Tx(2),Tn(2)]);

    Dubin_circle(xcf,ycf,radius,Tn, fin_pos);


    h1 = plot(Tx(1),Tx(2),'x','MarkerSize',10);%
    h2 = plot(Tn(1),Tn(2),'*','MarkerSize',10);
    h3 = plot(int_pos(1),int_pos(2),'s','MarkerSize',10);


end

%Function to plot the Dubin circles
function [h,h4,ang2] = Dubin_circle(x,y,r,in_pnt,fin_pnt)
    hold on
    ang1 = atan2d((in_pnt(2)-y),(in_pnt(1)-x));
    ang2 = atan2d((fin_pnt(2)-y),(fin_pnt(1)-x));

    th = 0:pi/50:2*pi;
    th1 = ang1:0.1:ang2;
    xunit = r * cos(th) + x;
    yunit = r * sin(th) + y;
    xu = r * cosd(th1) + x;
    yu = r * sind(th1) + y;
    h4 = plot(xunit, yunit,'r');
    h = plot(xu,yu,'b');%

end
```

13.4. APPENDIX-4

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                    %
%   Dubins Path Planning                             %
%                                                    %
%                                                    %
%                  Created by Balasekhar CSK         %
%                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [xcs,xcf,ycs,ycf, Tx,Tn,phi_ex,phi_en] = Dubins_Path(int_pos, way_pnt,
radius)

    % initial circle centre x and y value
    xcs = (int_pos(1,1) + radius* cos(int_pos(1,3)- pi/2));
    ycs =  (int_pos(1,2) + radius* sin(int_pos(1,3)- pi/2));

    %final circle centre x and y value
    xcf = (way_pnt(1,1) + radius* cos(way_pnt(1,3)- pi/2));
    ycf =  (way_pnt(1,2) + radius* sin(way_pnt(1,3)- pi/2)) ;

    % distance between centres of two circles
    c= sqrt( (xcf-xcs)^2+ (ycf-ycs)^2 );

    % psi is ignored since it is always 0 as initial and final radius is same
    phi_e= atan2((ycf- ycs),(xcf- xcs));

    % exit and entry angle
    phi_ex= phi_e+pi/2;
    phi_en= phi_e+pi/2;

    % exit and entry Point
    Tx= [xcs + radius* cos(phi_ex ), ycs + radius* sin(phi_ex)];
    Tn= [xcf + radius* cos(phi_en), ycf + radius* sin(phi_en)];
end
```

13.5. APPENDIX-5

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                    %
%   Carrot Chasing Algorithm                         %
%                                                    %
%                                                    %
%                  Created by Balasekhar CSK         %
%                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clc;
clear all; %#ok<CLALL>
close all;

% starting position
int_pos = [0,0,0];

% list of waypoints in the format [x,y,heading angle]
way_pnt = [0,10,0;
        60,60,pi/4;
        80,120,pi/6;
        150,70,(-pi/2);
        100,30,(-pi/3);
        50,0,-pi];

radius = 5;
radius1 = 5;
phi = 0;

% to estimate for all the waypoints
for i = 1:length(way_pnt)

    %Initialising the first point from where UGV will Travel
    xi = int_pos(1,1); yi = int_pos(1,2); theta_s = int_pos(1,3);

    %Initialising the final point of reach
    xf = way_pnt(i,1); yf = way_pnt(i,2); theta_w = way_pnt(i,3);
    way= way_pnt(i,:);

    % Dubins_Path is user defined function for dubins which returns
    % constraints
    [xcs,xcf,ycs,ycf,Tx,Tn] = Dubins_Path(int_pos,way,radius);

    %Plot the path estimated by Dubin algorithm
    [h,h1,h2,h3,h4,thetad] =
plotter(xcs,xcf,ycs,ycf,radius,Tx,Tn,[xi,yi],[xf,yf]);
    pause(0.5);


    pos(:,1) = [xi yi int_pos(1,3)].';

    %Define the First Dublin circle ie entry circle from initial way point
    [~,~] = CCA_Dubin_Circle([xcs,ycs],pos,radius1,thetad);
    pause(0.1);

    ang = atan2(Tn(2)-Tx(2), Tn(1)-Tx(1));

    %Define the path between 2 Dublin circles
    CCA_Straight_Path(Tx',Tn',[Tx ang]');
```

```
    pause(0.1);

    thetad = atan2((way_pnt(i,2)-ycf),(way_pnt(i,1)-xcf));
    Tn = [Tn(1); Tn(2); ang];

    %Define the second Dublin circle ie exit circle to final way point
    [phi,h5] = CCA_Dubin_Circle([xcf,ycf],Tn,radius1,thetad);
    pause(0.1);

    int_pos = way_pnt(i,:);
    pause(0.5);
end

%legend
plot(int_pos(1),int_pos(2),'s','MarkerSize',8);


xx = 0;yy =10.5;
for j = 1:length(way_pnt)
    xx = [xx,way_pnt(j,1)];
    yy = [yy,(way_pnt(j,2)+0.5)];
end
labels = {"int pos","way pnt1","way pnt2","way pnt3","way pnt4","way pnt5","way
pnt6"}; %#ok<CLARRSTR>
text(xx,yy,labels,'VerticalAlignment','HorizontalAlignment','right');
legend([h h1 h2 h3 h4 h5], {"Expected Path", "Exit point","Entry point",
"waypoint","Dubins Circle",  "Tracked Path"}); %#ok<CLARRSTR>

hold off

%Funtion  to plot the path estimated by Dubin algorithm
function [h,h1,h2,h3,h4,thetad] = plotter(xcs,xcf,ycs,ycf,radius,
Tx,Tn,int_pos,fin_pos)

    [h,h4,thetad] = circle(xcs,ycs,radius,int_pos,Tx);
    hold on


    line([Tx(1),Tn(1)],[Tx(2),Tn(2)]);
    circle(xcf,ycf,radius,fin_pos, Tn);

    h1 = plot(Tx(1),Tx(2),'x','MarkerSize',10);%, 'DisplayName','Exit Point');
    h2 = plot(Tn(1),Tn(2),'*','MarkerSize',10);%, 'DisplayName','Entry Point');
    h3 = plot(int_pos(1),int_pos(2),'s','MarkerSize',10);%,'DisplayName','Point');
    %plot(fin_pos(1),fin_pos(2),'o','MarkerSize',10);

end

%Function to plot the Dubin circles
function [h,h4,ang2] = circle(x,y,r,in_pnt,fin_pnt)
    hold on
    ang1 = atan2((in_pnt(2)-y),(in_pnt(1)-x));
    ang2 = atan2((fin_pnt(2)-y),(fin_pnt(1)-x));

    th1 = linspace(ang1, ang2, 250);
    th = 0:pi/50:2*pi;
    xunit = r * cos(th) + x;
    yunit = r * sin(th) + y;
    xu = r * cos(th1) + x;
```

```
    yu = r * sin(th1) + y;
    h4 = plot(xunit, yunit,'r--');%,'DisplayName','Dubins Circle');
    h = plot(xu,yu,'b');%,'DisplayName','Path line');

end
```

13.6. APPENDIX-6

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                    %
%   CCA_Dublin Circle                                %
%                                                    %
%                                                    %
%                 Created by Balasekhar CSK          %
%                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [phi,h5] = CCA_Dubin_Circle(O,p,r,thetad)%
    i =  0 ;                 % Time Index

     x(1) = p(1,1);
     y(1) = p(2,1);

    psi(1) = p(3,1);
    % Angle Converting Parameters
    r2d                      =              180 / pi ;          % Radian to Degree
[-]
    d2r                      =              1 / r2d ;           % Degree to Radian
[-]

    % Carrot Distance
    lambda                   =              0.01 * d2r ;

    % Gain
    kappa                    =              2 ;
    tol                      =              3;
    dt                       =              0.001;

    va                       =              5 ;       % UGV velocity [m/s]
    umax                     =              5 ;       % UGV Maximum Lateral
Acceleration [m/s2]
    t                        =              0  ;
    phi                      =              0;

    while i<6000
        i =  i + 1 ;

        %.. Path Following Algorithm
        % Orientation of vector from initial waypoint to final waypoint, theta
        theta   =  atan2( p(2,i) - O(2), p(1,i) - O(1) ) ;

        del_THETA = theta-thetad;
        if abs(del_THETA)*r2d <= tol     % you can change the tol
            break
        end


        % Carrot position, s = ( xt, yt )
        xt  = O(1) + r * cos( theta + lambda ) ;
        yt  = O(2) + r * sin( theta + lambda ) ;


        % Desired heading angle, psid
        psid  = atan2( yt - y(i), xt - x(i) ) ;

        % Heading angle error, DEL_psi
```

```
        DEL_psi = psid - psi(i) ;
        % Wrapping up DEL_psi
        DEL_psi  =  rem(DEL_psi,2*pi);

        %wrapping DEL_psi within pi
        if DEL_psi < -pi
            DEL_psi = DEL_psi + 2*pi;
        elseif DEL_psi > pi
            DEL_psi = DEL_psi-2*pi;
        end


        % Guidance command, u
        u(i)   =  kappa * DEL_psi * va ;
        % Limit u
        if u(i) > umax
            u(i) = umax;

        elseif u(i) < -umax
            u(i)  =  - umax;
        end

        %.. UGV Dynamics
        % Dynamic Model of UGV
        dx   =  va * cos( psi(i) ) ;
        dy   =  va * sin( psi(i) ) ;
        dpsi = u(i) / va ;

        % UGV State Update
        x(i+1)  =  x(i) + dx * dt ;
        y(i+1)  =  y(i) + dy * dt ;
        psi(i+1)=  psi(i) + dpsi * dt ;
        phi     =  psi(i);
        % UGV Position Vector Update
        p(1:2,i+1) = [ x(i+1), y(i+1) ]' ;

        %.. Time Update
        t(i+1)  =  t(i) + dt ;
        %dis = norm
    end
    disp(t(i));
    h5 = plot( x, y,'k','LineWidth', 2) ;
end
```

13.7. APPENDIX-7

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%   CCA_Straight                                                                %
%                                                                               %
%                                                                               %
%                                                  Created by Balasekhar CSK %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [phi] = CCA_Straight_Path(Wi,Wf, p)

    i   =  0 ;                  % Time Index
    x(1) = p(1,1);
    y(1) = p(2,1);

    psi(1) = p(3,1);
    t(1)  =  0;

    va  = 5 ;                % UGV Velocity [m/s]
    umax   =  5;      % UGV Maximum Lateral Acceleration [m]

    %.. Design Parameters

    kappa   =   2 ;              % Gain

    %%%% Modify %%%%%%%%%%%%

    dt     =  0.001;
    delta  =  1e6 ;
    phi    =  0;

    if Wi(1)< Wf(1)
        while (x(i+1) <Wf(1))% to draw line till we reach final point
            i   =  i + 1 ;


%=============================================================================%
            %.. Path Following Algorithm

            % Step 1
            % Distance between initial waypoint and current UGV position, Ru
            Ru    =  norm( p(1:2,i) - Wi, 2 ) ;
            % Orientation of vector from initial waypoint to final waypoint, theta
            theta =  atan2( Wf(2) - Wi(2), Wf(1) - Wi(1) ) ;

            % Step 2
            % Orientation of vector from initial waypoint to current UGV position,
thetau
            thetau   =  atan2( p(2,i) - Wi(2), p(1,i) - Wi(1) ) ;
            % Difference between theta and theatu, DEL_theta
            DEL_theta = theta - thetau ;

            % Step 3
            % Distance between initial waypoint and q, R
            R   =  sqrt( Ru^2 - ( Ru * sin( DEL_theta ) )^2 ) ;

            % Step 4
```

```
            % Carrot position, s = ( xt, yt )
            xt  =  ( R + delta ) * cos( theta ) ;
            yt  =  ( R + delta ) * sin( theta ) ;

            % Step 5
            % Desired heading angle, psid
            psid   =  atan2( yt - y(i), xt - x(i) ) ;
            % Wrapping up psid within pi
            psid = rem(psid,2*pi);
            if psid < -pi
                psid = psid + 2*pi;
            elseif psid > pi
                psid = psid-2*pi;
            end

            % Step6
            % Guidance command, u
            u(i)    =   kappa * ( psid - psi(i) ) * va ;
            % Limit u
            if u(i) > umax
                u(i) =  umax;
            elseif u(i) < -umax
                u(i) = - umax;
            end

            % UGV Dynamics
            % Dynamic Model of UGV
            dx   =  va * cos( psi(i) ) ;
            dy   =   va * sin( psi(i) ) ;
            dpsi  =   u(i) / va ;

            % UGV State Update
            x(i+1)  =  x(i) + dx * dt ;
            y(i+1)  =  y(i) + dy * dt ;
            psi(i+1)  =  psi(i) + dpsi * dt ;
            phi   =  psi(i);

            % UGV Position Vector Update
            p(1:2,i+1)  = [ x(i+1), y(i+1) ]' ;

            %.. Time Update
            t(i+1)   =  t(i) + dt ;

        end



    else
        while (x(i+1) >Wf(1)) %reverse condition when vehicle moves towards the
circle
            i    =  i + 1 ;


            % Path Following Algorithm

            % Step 1
            % Distance between initial waypoint and current UGV position, Ru
            Ru  =  norm( p(1:2,i) - Wi, 2 ) ;
            % Orientation of vector from initial waypoint to final waypoint, theta
```

```
            theta =  atan2( Wf(2) - Wi(2), Wf(1) - Wi(1) ) ;

            % Step 2
            % Orientation of vector from initial waypoint to current UGV position,
thetau
            thetau  =  atan2( p(2,i) - Wi(2), p(1,i) - Wi(1) ) ;
            % Difference between theta and theatu, DEL_theta
            DEL_theta  =  theta - thetau ;

            % Step 3
            % Distance between initial waypoint and q, R
            R  =  sqrt( Ru^2 - ( Ru * sin( DEL_theta ) )^2 ) ;

            % Step 4
            % Carrot position, s = ( xt, yt )
            xt   =  ( R + delta ) * cos( theta ) ;
            yt   =  ( R + delta ) * sin( theta ) ;

            % Step 5
            % Desired heading angle, psid
            psid     =    atan2( yt - y(i), xt - x(i) ) ;
            % Wrapping up psid to pi
            psid = rem(psid,2*pi);
            if psid < -pi
                psid = psid + 2*pi;
            elseif psid > pi
                psid = psid-2*pi;
            end

            % Step6
            % Guidance command, u
            u(i)  =  kappa * ( psid - psi(i) ) * va ;

            % Limit u
            if u(i) > umax
                u(i)  =  umax;
            elseif u(i) < -umax
                u(i)  =  - umax;
            end

            %.. UGV Dynamics
            % Dynamic Model of UGV
            dx     =  va * cos( psi(i) ) ;
            dy     =  va * sin( psi(i) ) ;
            dpsi   =  u(i) / va ;

            % UGV State Update
            x(i+1)    =   x(i) + dx * dt ;
            y(i+1)    =   y(i) + dy * dt ;
            psi(i+1)  =   psi(i) + dpsi * dt ;
            phi       =   psi(i);
            % UGV Position Vector Update
            p(1:2,i+1)  =  [ x(i+1), y(i+1) ]' ;

            %.. Time Update
            t(i+1)  = t(i) + dt ;

        end
```

```
    end

    plot( x, y,'k', 'LineWidth', 2 ) ;% plot function

end
```