16/3/2019

Motion Control of a Mobile Robot

ME425 HW1

EDIN GUSO
23435

Contents (pg.1)

Introduction

The goal of this homework assignment was to design and simulate the position control of a nonholonomic, differential drive wheeled mobile robot. The task consisted of designing the Simulink model of the kinematic model and the motion control of the robot, writing the required MATLAB codes and finally simulating the system with different control parameters and initial conditions.

Procedure

Since the kinematic model and the motion control algorithm can be complicated to model in Simulink, I had to use a systematic method while building the Simulink model. I have divided the task into three main parts. The first part was the kinematic model of the robot. The second part was the motion control algorithm. Finally, the last part was the polar coordinate to cartesian coordinate transformation algorithm which allowed me to plot the robot position in the x-y plane.

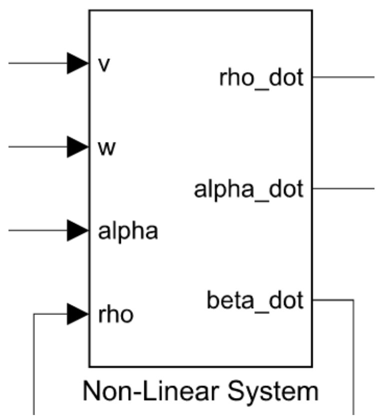


Figure 2.a Kinematic Model Block

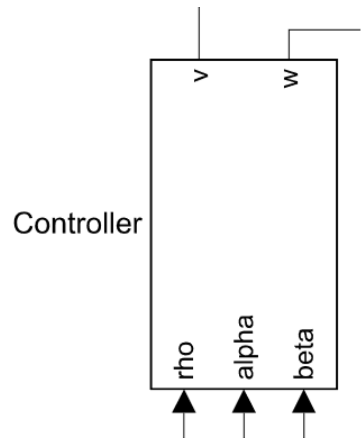


Figure 2.b Motion Control Block

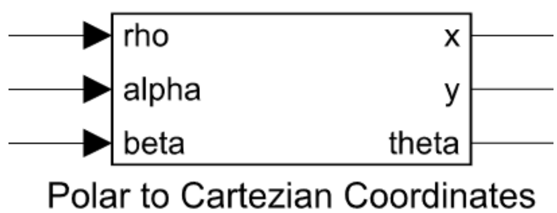


Figure 2.c Coordinate Transformation Block

Breaking up the task into smaller parts has made the Simulink model look much cleaner and made it easier to spot and correct mistakes.

Also, in order to see several movements from different initial positions, I have used a for loop in my .m file and simulated the Simulink model with the sim command.

Related Equations and Stability Analysis

The equations which have been used for modelling the non-linear taken from our reference book can be seen below. There are two equations modelled for the goal being in front of the robot ($\alpha \in \mathrm{I}_1$)and behind the robot ($\alpha \in \mathrm{I}_2$). This is done in order to have backwards motion of the motor the same motion as if the robot moving forwards:

$$\begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} = \begin{bmatrix} -\cos\alpha & 0 \\ \dfrac{\sin\alpha}{\rho} & -1 \\ -\dfrac{\sin\alpha}{\rho} & 0 \end{bmatrix} \begin{bmatrix} \upsilon \\ \omega \end{bmatrix} \qquad for\ I_1 = (\frac{-\pi}{2}, \frac{\pi}{2}]$$

$$\begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} = \begin{bmatrix} \cos\alpha & 0 \\ -\dfrac{\sin\alpha}{\rho} & -1 \\ \dfrac{\sin\alpha}{\rho} & 0 \end{bmatrix} \begin{bmatrix} \upsilon \\ \omega \end{bmatrix} \qquad for\ I_2 = (-\pi, -\frac{\pi}{2}] \cup (\frac{\pi}{2}, \pi]$$

Similarly, the control law equations which have been taken from our reference book are listed:

$$\upsilon = k_\rho \rho \qquad \omega = k_\alpha \alpha + k_\beta \beta$$

The stability criterion of this system is determined locally by the Lyapunov's stability theorem. This theorem states that if we linearize a non-linear system and prove its stability, it assures that the non-linear system is locally stable, as it is shown in the lecture slides:

Proof:
for small x –> $\cos x = 1$, $\sin x = x$

$$\begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} = \begin{bmatrix} -k_\rho & 0 & 0 \\ 0 & -(k_\alpha - k_\rho) & -k_\beta \\ 0 & -k_\rho & 0 \end{bmatrix} \begin{bmatrix} \rho \\ \alpha \\ \beta \end{bmatrix} \qquad A = \begin{bmatrix} -k_\rho & 0 & 0 \\ 0 & -(k_\alpha - k_\rho) & -k_\beta \\ 0 & -k_\rho & 0 \end{bmatrix}$$

and the characteristic polynomial of the matrix A of all roots

$$(\lambda + k_\rho)(\lambda^2 + \lambda(k_\alpha - k_\rho) - k_\rho k_\beta)$$

have negative real parts.

The control gains that assure local stability can be listed as below, as it can be seen in the lecture slides:

$$k_\rho > 0 \ ; \ k_\beta < 0 \ ; \ k_\alpha - k_\rho > 0$$

Results

In order to have a basis case to compare my results to, I have decided to use the control parameters given in our textbook which are k_rho = 3, k_alpha = 8, k_beta = -1,5. Simulating my system with these values and tweaking the parameters one by one has made it much easier to understand the individual effects of each parameter. In each test, I have simulated 8 robots, 45 degrees apart, at distance rho.

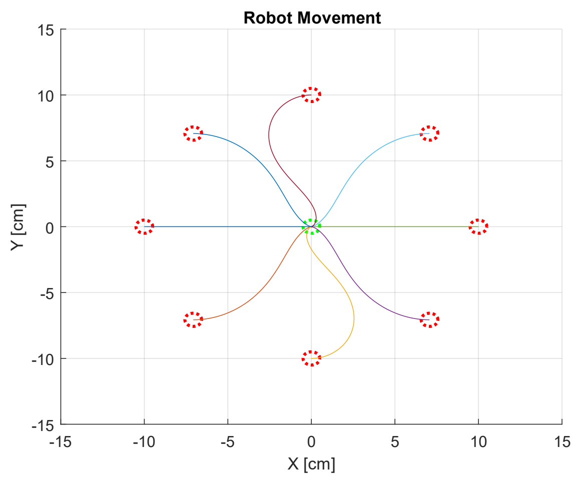


Figure 3.a All Robots Facing Right

The first simulation I did was with the previously mentioned control parameters, with robots' starting positions 10cm away from the goal and all of them facing right (+x direction). The results can be seen in the figure 3.a. The results were as expected as the robots changed their angles and moved forward to adjust their x-y positions. When they got closer to the goal, they adjusted their angle. This was true for both the robots facing the goal and the ones facing away from the goal. This was as expected because the controller I had implemented makes the robots facing away from the goal do the same motion as if they were facing towards the goal, just inverting their linear speed.

After that experiment, I kept all the control gains and starting positions the same, and only changed the starting angles. I have set each robot's initial angle to make them look directly at the goal. The results can be seen in figure 3.b. The results were very unexpected. I expected each robot to change their angle slightly to take the shortest path but the robots in the top half have surprisingly taken the longer path. All the robots preferred to approach the goal from below.

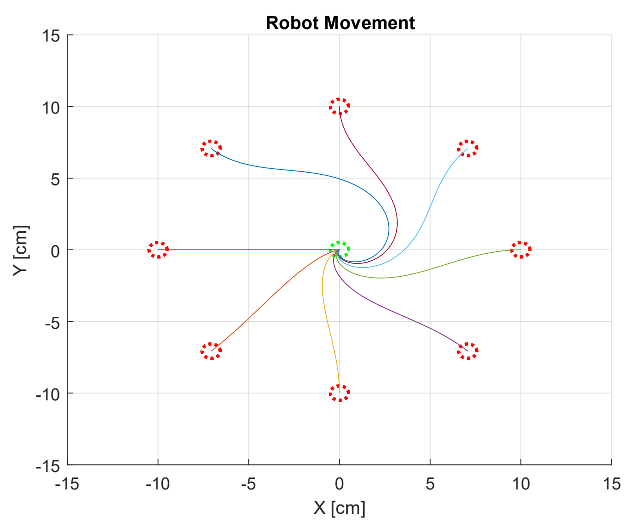


Figure 3.b All Robots Facing the Goal

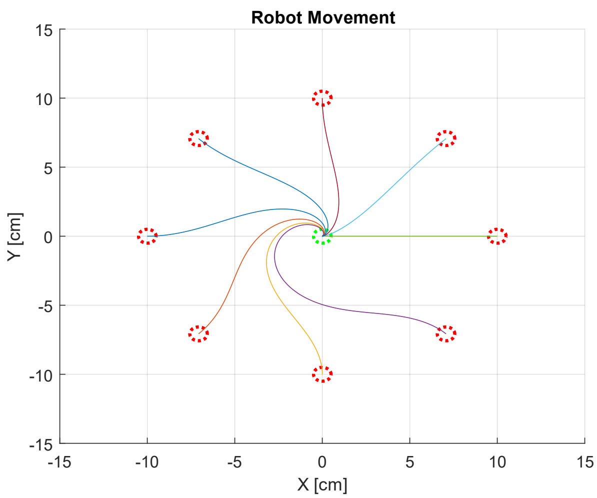


Figure 3.c All Robots Facing away from the Goal

My next question was what a group of robots facing the goal backwards would do. My expectation would have been that all of them would go backwards towards the goal with the shortest path if I hadn't seen the results in figure 3.b. But since some of them preferred to take the longer path, my guess is that these robots would do the same thing. It can be seen in figure 3.c that my guess was correct, and all of the robots preferred approaching the goal from the upper side (it was from the bottom in forward case).

After finishing the tests about the starting angles' effect on motion control, I decided to test the effects of control gains on motion control. Just by observing the equations, it is easy to tell that k_rho is related to the linear velocity and that k_alpha and k _beta are related to angular velocity. In order to see how the increase in k_rho would affect the movement, I kept all the initial values the same, only changed k_rho from 3 to 7. Also, I set all the robots' initial directions facing right in order to observe both forward and backward motion. As it can be seen from figure 3.d, the robots aggressively move towards the goal which is caused by high linear velocity but have to slow down and correct their angles a lot towards the goal position.

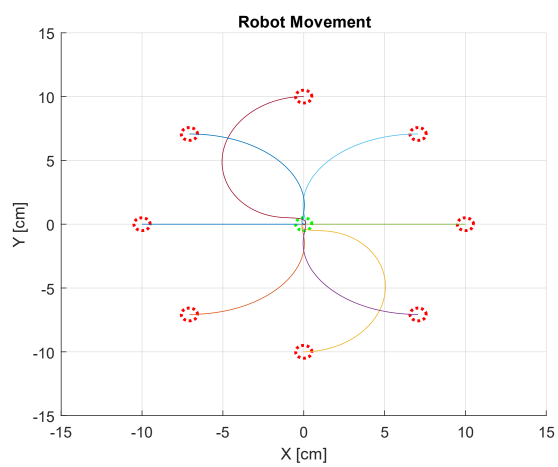


Figure 3.d All Robots Facing Right, large k_rho

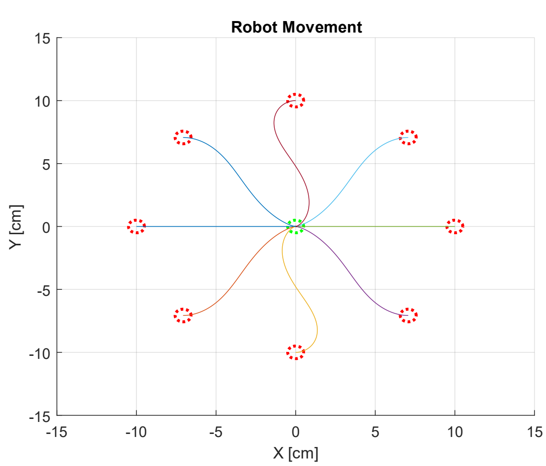


Figure 3.e All Robots Facing Right, large k_alpha and k_beta

Then I did the exact same experiment, the only difference being that I kept k_rho constant and increased k_alpha to 15 and k_beta to -4. Observing the results from the previous experiment, I expected the robots to move slowly and accurately towards the goal. As it can be seen in figure 3.e, my expectations were correct, and the robots were arriving slowly with minimal angle error to the goal position.

After understanding the effects of control gains on the motion control, I also wanted to test the effects of distance to the goal (rho). In all my other tests, the rho used was 10cm. In this test, I decided to use 50cm and once again, with the facing right configuration to observe different motions. The results were interesting. As it can be seen in the figure 3.f, it is the exact similar results as in figure 3.a. I have realized that no matter how much I increase or decrease the rho, the results will be the same (I have tested very high rho values such as 10^6, etc.). The reason behind that is there are no actuator limits in the Simulink environment.

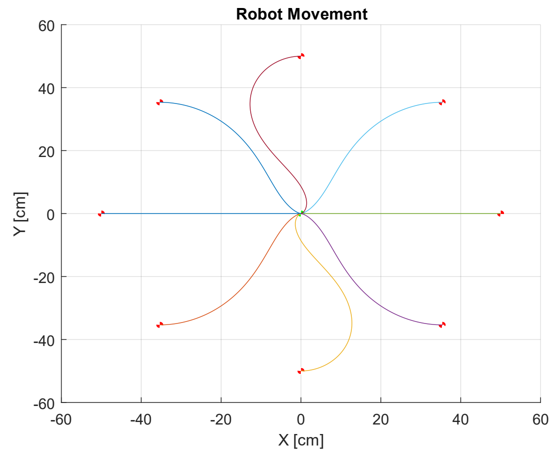


Figure 3.f All Robots Facing Right, large rho

Conclusion

After finishing all the different tests with different initial conditions and control gains, I have come up with 4 main conclusions.

Robots starting from facing goal and facing away from goal have the exact same motion. It can be seen from figure 3.a that the figure has a symmetry line between the motion of robots moving forwards and robots moving backwards.

Robots do not necessarily take the shortest path. As it can be proven with the figure 3.b and figure 3.c, when this control law is applied, the robots do not always follow the shortest path there is towards the goal. That was not what this control law promised anyway, the statement in our lecture slides was that “parking maneuver is performed always in the most natural way”.

Control gains heavily affect the motion of the robots. Robots tend to approach the goal faster with more angle error, having to adjust a lot when they get closer to the goal if k_rho is increased. On the other hand, robots move much slower and approach the goal with minimum angle error if k_alpha and k_beta values are high. The effects of control gains are available in figures 3.d and 3.e.

Finally, the initial distance to the goal (rho_init) does not affect the performance or the stability of the robot at all in the Simulink environment. The reason behind that is that there are no actuator constraints in Simulink. However, in real world the case would be very different. Therefore, we can conclude that for larger rho_init values, the Simulink results are not very accurate.

The problem of motion control of nonholonomic, differential drive wheeled mobile robots is very complicated. But dividing the problem into smaller pieces and then linking them together with logical algorithms makes the problem possible to solve.

Appendix

ME425_HW1.m

```
clear all; clc;

%Controller Value Initialization
k_rho = 3;      %k_rho > 0
k_alpha = 8;    %k_alpha > k_rho
k_beta = -1.5; %k_beta < 0

fig = figure('Name','Robot Movement');
hold on;

%Robot outline radius
cri = 0.5;

%Initial distance to goal
ri = 10;

%Draw the goal robot outline
viscircles([0 0], cri, 'Color', 'g', 'LineStyle', ':');
for i = 1:8
    %Robot position initialiation, converting them into polar coordinates
    %and drawing the initial robot outline for each case
    switch i
        case 1
            xi = -ri;
            yi = 0;
            ti = 0;
            viscircles([xi yi], cri, 'Color', 'r', 'LineStyle', ':');
            [rho_init, alpha_init, beta_init] = initialization(xi, yi, ti,
1);
        case 2
            xi = -ri/(2^0.5);
            yi = -ri/(2^0.5);
            ti = 0;
            viscircles([xi yi], cri, 'Color', 'r', 'LineStyle', ':');
            [rho_init, alpha_init, beta_init] = initialization(xi, yi, ti,
1);
        case 3
            xi = 0;
            yi = -ri;
            ti = 0;
            viscircles([xi yi], cri, 'Color', 'r', 'LineStyle', ':');
            [rho_init, alpha_init, beta_init] = initialization(xi, yi, ti,
1);
```

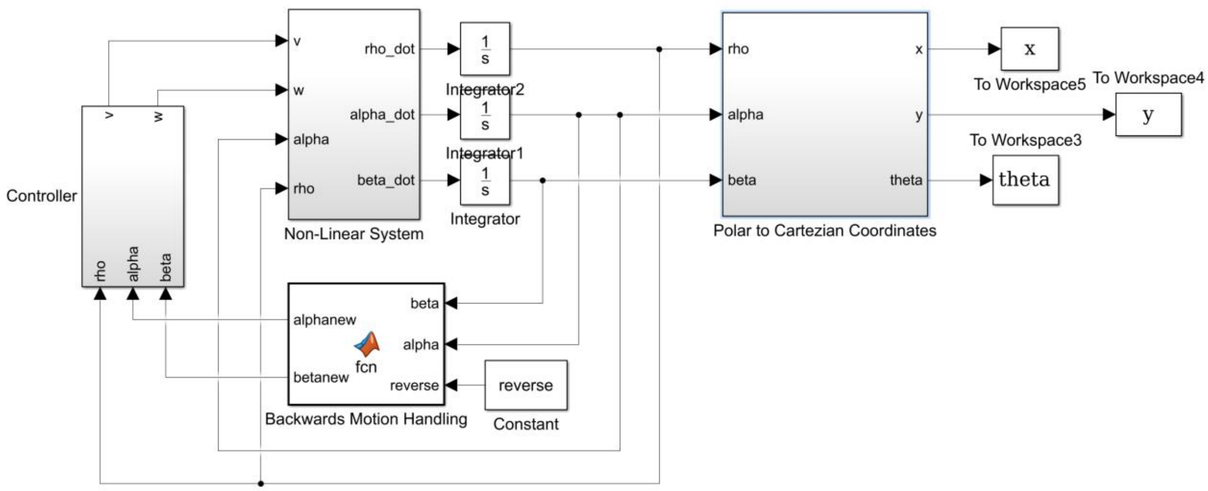
ME425_HW1.m

```
        case 4
            xi = ri/(2^0.5);
            yi = -ri/(2^0.5);
            ti = 0;
            viscircles([xi yi], cri, 'Color', 'r', 'LineStyle', ':');
            [rho_init, alpha_init, beta_init] = initialization(xi, yi, ti,
1);
        case 5
            xi = ri;
            yi = 0;
            ti = 0;
            viscircles([xi yi], cri, 'Color', 'r', 'LineStyle', ':');
            [rho_init, alpha_init, beta_init] = initialization(xi, yi, ti,
1);
        case 6
            xi = ri/(2^0.5);
            yi = ri/(2^0.5);
            ti = 0;
            viscircles([xi yi], cri, 'Color', 'r', 'LineStyle', ':');
            [rho_init, alpha_init, beta_init] = initialization(xi, yi, ti,
1);
        case 7
            xi = 0;
            yi = ri;
            ti = 0;
            viscircles([xi yi], cri, 'Color', 'r', 'LineStyle', ':');
            [rho_init, alpha_init, beta_init] = initialization(xi, yi, ti,
1);
        case 8
            xi = -ri/(2^0.5);
            yi = ri/(2^0.5);
            ti = 0;
            viscircles([xi yi], cri, 'Color', 'r', 'LineStyle', ':');
            [rho_init, alpha_init, beta_init] = initialization(xi, yi, ti,
1);
    end
    reverse = 0;
    if (alpha_init <= -pi/2 || alpha_init > pi/2)
       reverse = 1;
    end

    %simulating the system
    sim('KinematicModel');
    %plotting the results
    plotData(x.Data, y.Data);
end
hold off;
%save the obtained figure as an svg file
saveas(fig, 'RobotMovement.svg');
```

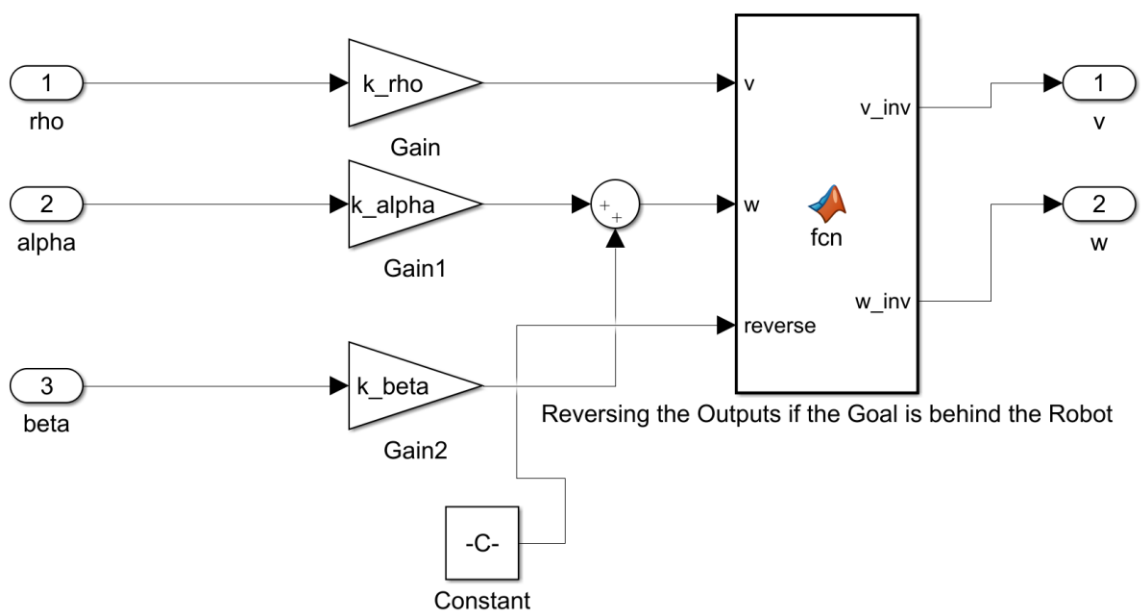
ME425_HW1.m

```
%plots the x - y data
function plotData(x, y)
plot(x,y);
xlabel('X [cm]'); ylabel('Y [cm]'); grid on;
title('Robot Movement');
end

%checks whether the angle is in the allowable range
function newAngle = angleChecker(angle)
while angle > pi
    angle = angle - pi;
end
while angle <= -pi
    angle = angle + pi;
end
newAngle = angle;
end

%if the last parameter is 0, accepts the coordinates as polar,
%else calls the function for converting into polar
function [rho, alpha, beta] = initialization(param1, param2, param3,
polarCoordinate)
if(polarCoordinate == 0)
    rho = param1;
    alpha = angleChecker(param2);
    beta = angleChecker(param3);
else
    [rho, alpha, beta] = cartezianToPolar(param1, param2, param3);
end
end

%converts the entered cartezian coordinates into polar coordinates
function [rho, alpha, beta] = cartezianToPolar(x, y, theta)
deltaX = 0 - x;
deltaY = 0 - y;
rho = (deltaX^2 + deltaY^2)^(1/2);
alpha = angleChecker(-theta + atan2(deltaY,deltaX));
beta = -theta - alpha;
end
```

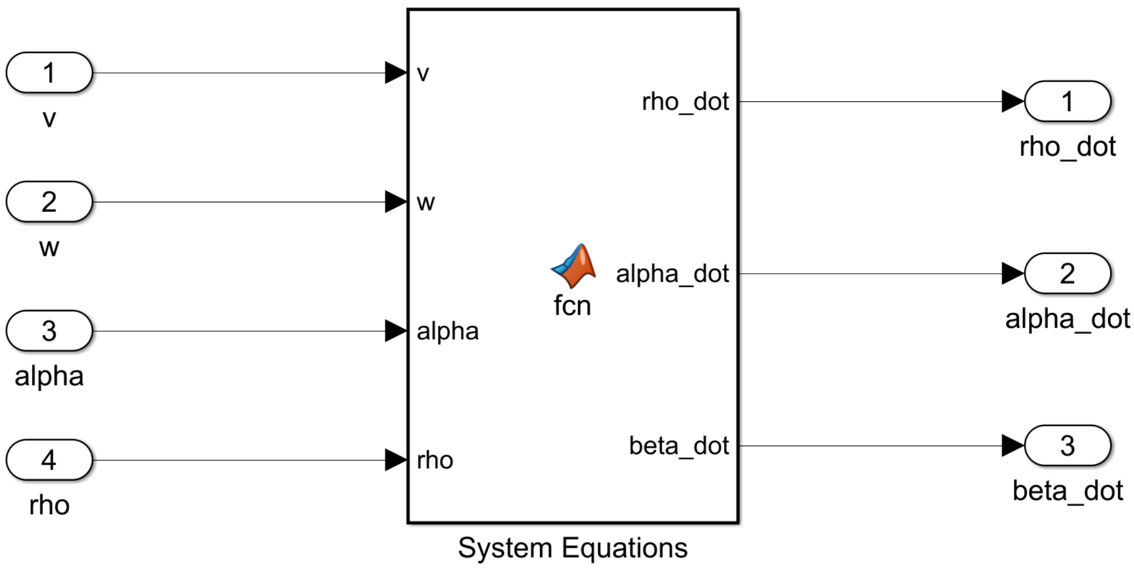
KinematicModel.slx



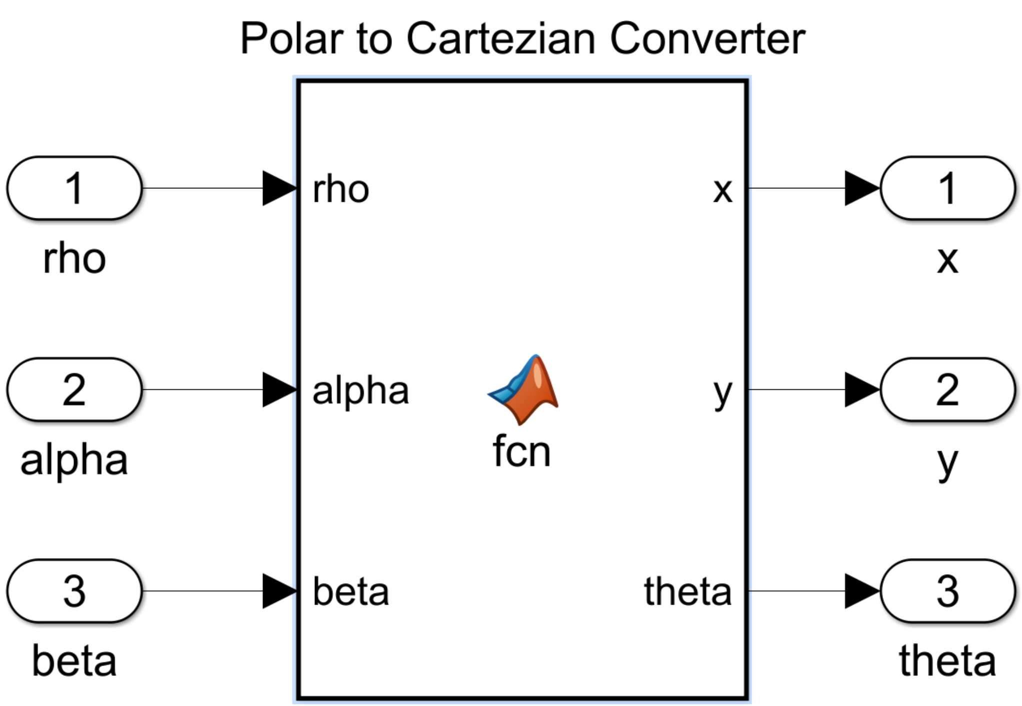
Subsystem: Controller



Subsystem: Non-Linear System



Subsystem: Polar to Cartezian Coordinates



Backwards Motion Handling

```
function [alphanew, betanew] = fcn(beta, alpha ,reverse)
alphanew = alpha;
betanew = beta;
if reverse
    alphanew = alphanew - pi;
    betanew = betanew + pi;
end
if (alphanew <= -pi)
   alphanew = alphanew + 2*pi;
end
if (betanew > pi)
    betanew = betanew - 2*pi;
end
```

Reversing the Outputs if the Goal is behind the Robot

```
function [v_inv,w_inv] = fcn(v, w, reverse)
w_inv = w;

if (reverse)
    v_inv = -v;
else
    v_inv = v;

end
```

System Equations

```
function [rho_dot, alpha_dot, beta_dot] = fcn(v,w, alpha, rho)
M = [-cos(alpha), 0;...
    sin(alpha)/rho, -1;...
    -sin(alpha)/rho, 0];

y = M*[v; w];

rho_dot = y(1);
alpha_dot = y(2);
beta_dot = y(3);
```

Polar to Cartezian Converter

```
function [x, y, theta] = fcn(rho, alpha, beta)
x = -rho*cos(-beta);
y = -rho*sin(-beta);
theta = -beta-alpha;
```