Table of Contents

Formatting

```
clc
clear all
close all
format compact
```

Robot characteristics

```
robot.X = 0;
robot.Y = 10;
robot.Phi = 0;
robot.radius = 1;
robot.width = 2;
robot.length = 1;
robot.Vel = 1;
robot.angVel = 0;
```

Wheel Characteristics

```
Wheel.radius = .25;
Wheel.wheel_width = 0.125;

Wheel.gamma = 0;
```

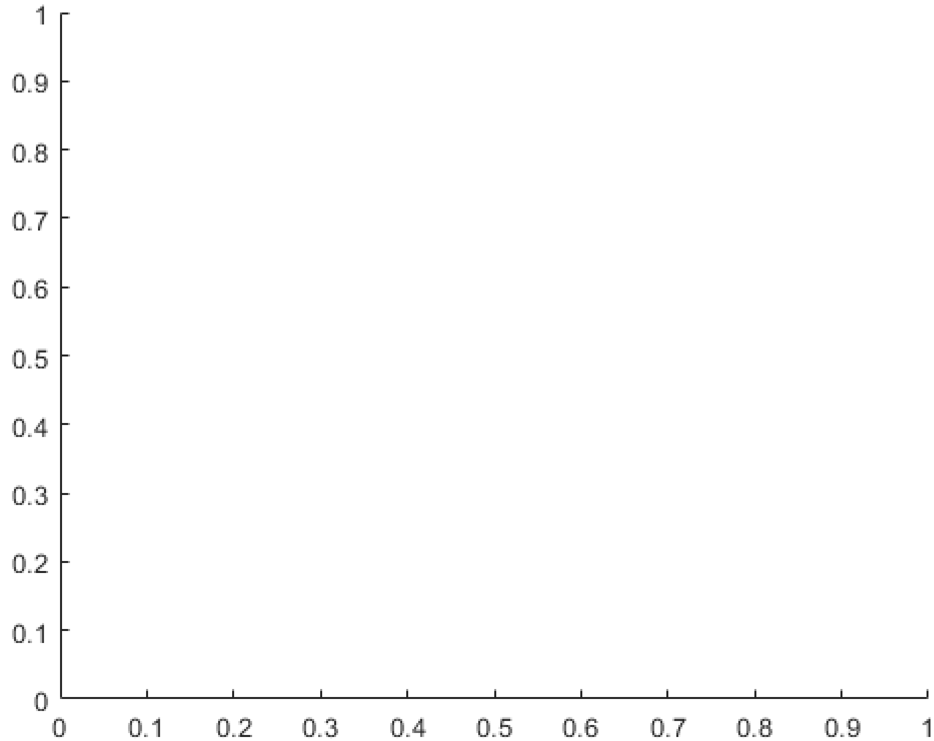
Control Inputs

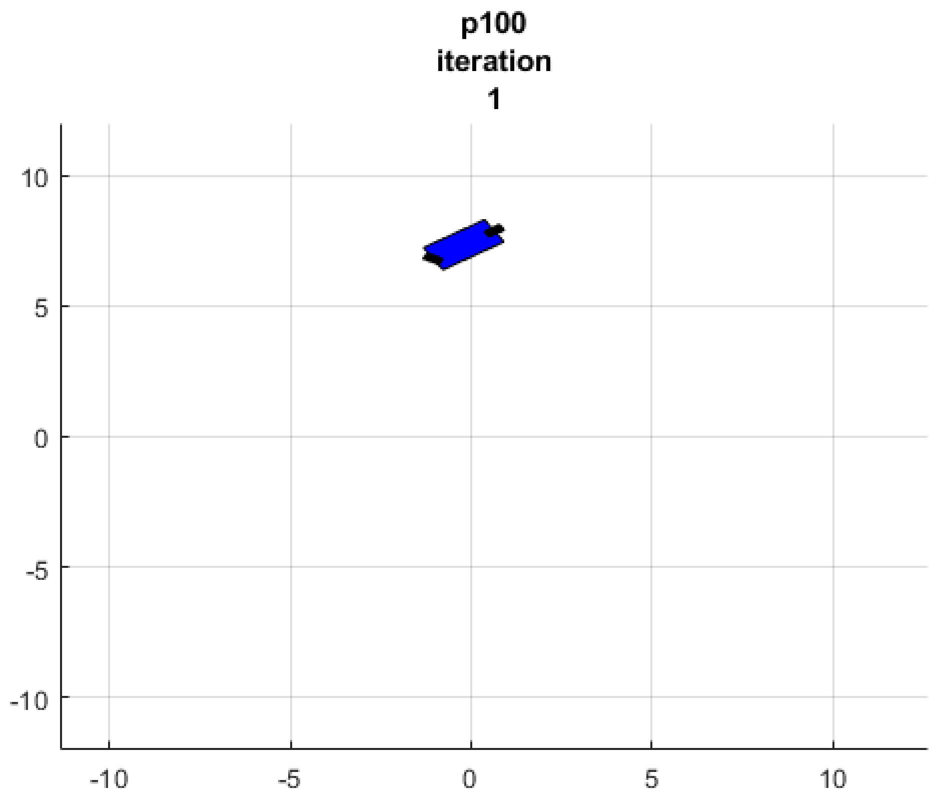
```
des.Y = 0;
old_error = 5;

InitialRobot = robot;
```

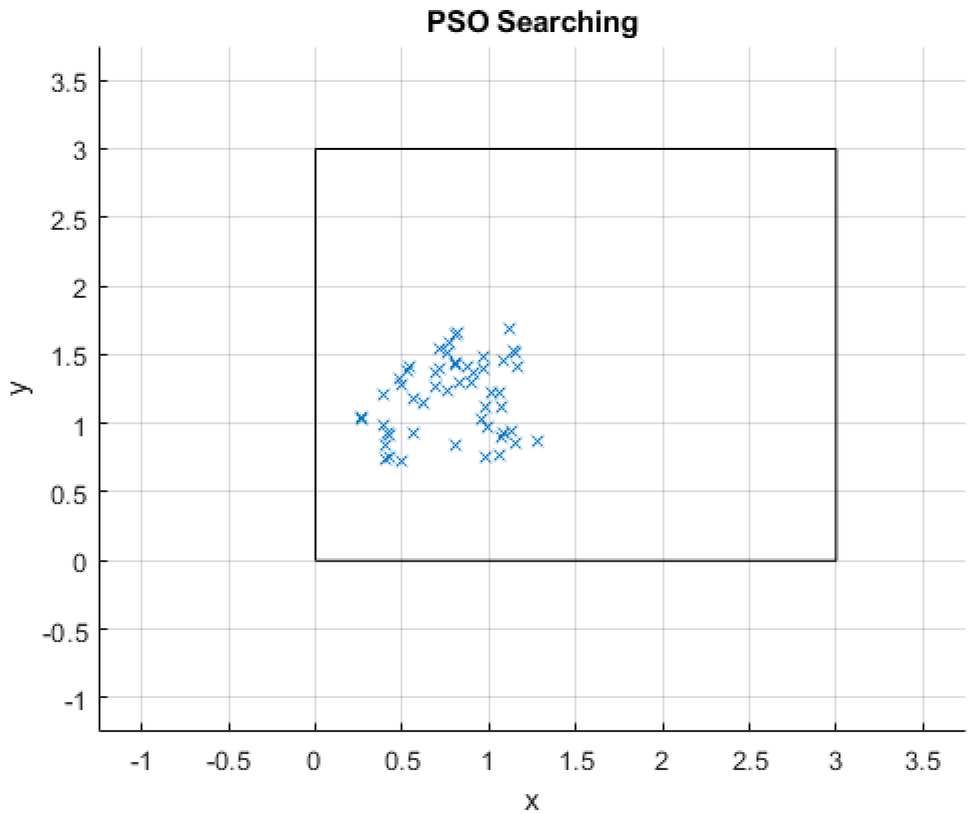
Passing to PSO

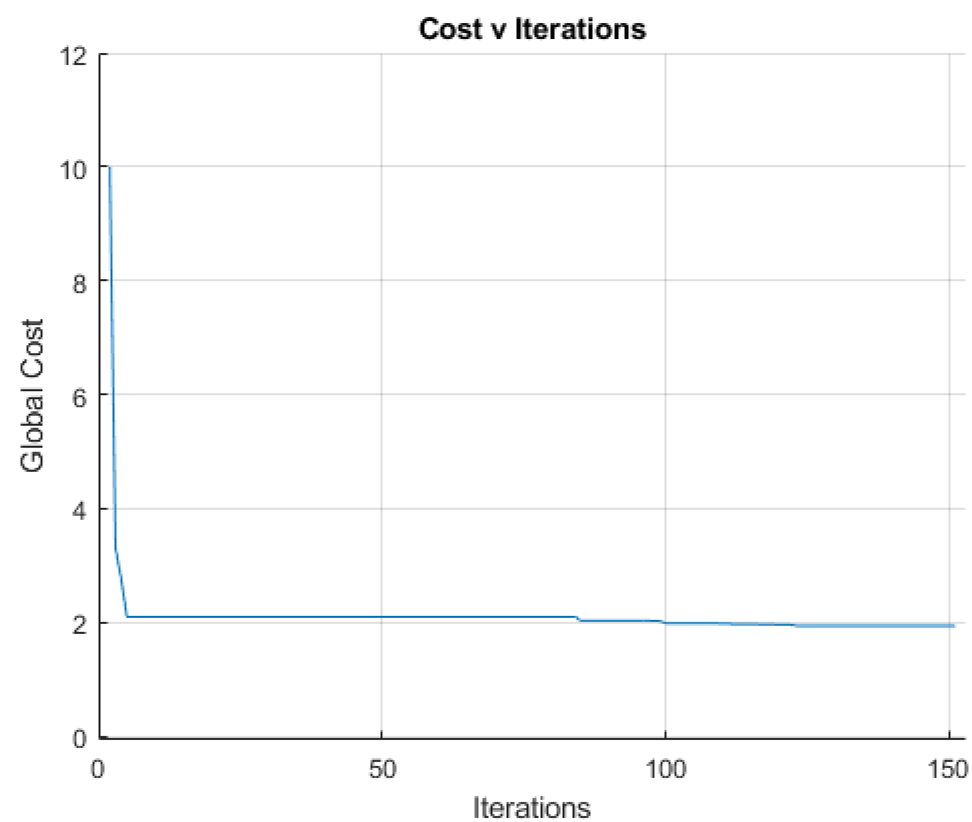
```
G = my_PSO(InitialRobot, des, old_error, Wheel);
```

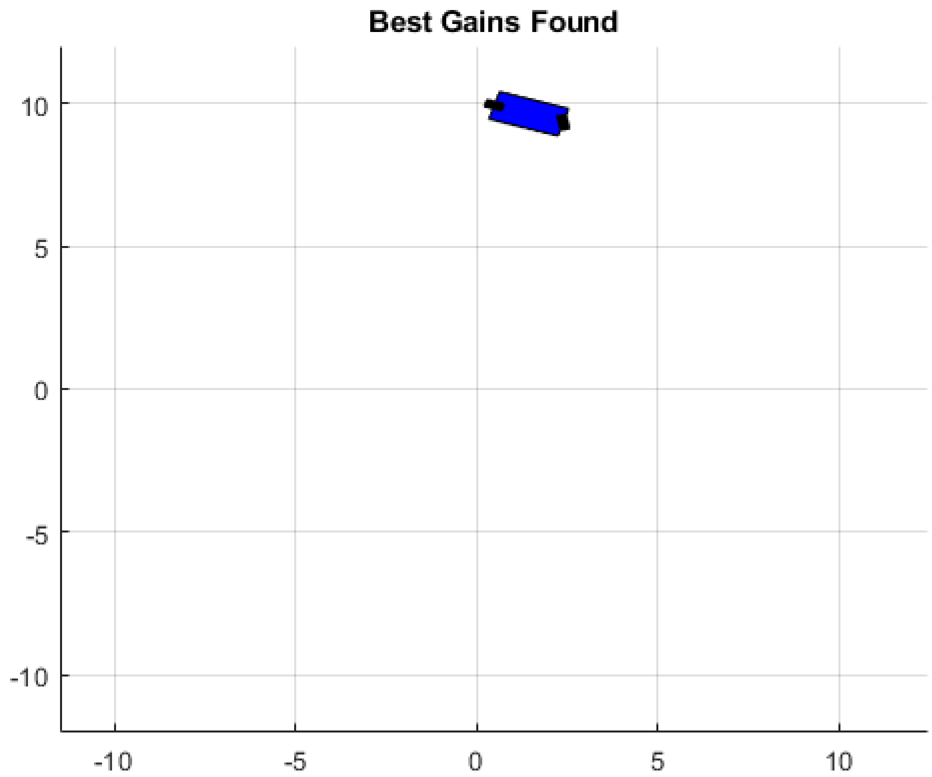
```
Best cost of 1.93 found at (0.27, 1.03)
```


1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


p100
iteration
1
10
5
0
-5
-10
-10
-5
0
5
10


PSO Searching
3.5
3
2.5
2
1.5
1
0.5
0
-0.5
-1
y
-1
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
x


Cost v Iterations
12
10
8
6
4
2
0
Global Cost
0
50
100
150
Iterations




```
function [Params] = PSOParams()

%Number of Particles
Params.N = 100;

%Search Boundaries
Params.ya = 0;
Params.yb = 3;
Params.xa = 0;
Params.xb = 3;

%Initializing particl name matrix
Params.paricle_names = zeros(1, Params.N);

%Initializing global values
Params.G.best_cost = 10;
Params.G.best_pos = [0 0];

%Setting alpha values, and alpha1 max and min
Params.alpha_1_max = 10;
Params.alpha_1_min = 1;
Params.alpha_1_delta = 0.5; %Chnage in alpha 1 each try

Params.alpha_2 = 3;
Params.alpha_3 = 6.5;
```

```
%Setting time variables (seconds)
Params.dt = 0.01;
Params.max_time = 2;

%Setting maximum velocity
Params.max_vel = 5;

%Initializing global cost matrix
Params.global_cost(1) = Params.G.best_cost;

%Plotting
Params.plotBool = 1; %1 for plots, 0 for no plots
Params.plotResolution = 50;%How many frames to advance from the previous one

end



function [G] = my_PSO(InitialRobot, des, old_error, InitialWheel)

%% Paramater Initialization
Params = PSOParams();

%% PSO Iterations

%Generating initial positions and velcities
for j = 1:Params.N

    % Random initial position
    rand_x = Params.xa + (Params.xb-Params.xa)*rand(1);
    rand_y = Params.ya + (Params.yb-Params.ya)*rand(1);

    % Creating each particle name
    particle_name = string(strcat('p', num2str(j)));
    particle_names(j) = particle_name;

    % Random Velocity
    p.(particle_name).vel = [2*rand(1) - 1, 2* rand(1) - 1];

    % Assigning random position to the particle
    p.(particle_name).pos = [rand_x, rand_y];

    % Arbitrarily assigning personal best for the particle
    p.(particle_name).best_cost = 1000;
    p.(particle_name).best_pos = p.(particle_name).pos;

    % Storing these positions in an array
    positions(j, :, 1) = p.(particle_name).pos; %third dimension is time

end

%% Picking variables out of the Params struct
%Global vals
```

```
global_cost = Params.global_cost;
G.best_cost = Params.G.best_cost;
G.best_pos = Params.G.best_pos;

%Time vars
dt = Params.dt;
max_time = Params.max_time;

%Alphas
alpha_1_max = Params.alpha_1_max;
alpha_1_delta = Params.alpha_1_delta;
alpha_1_min = Params.alpha_1_min;
alpha_2 = Params.alpha_2;
alpha_3 = Params.alpha_3;
max_vel = Params.max_vel;

%Number of particles
N = Params.N;

%% Values that won't change w/ dif simulations

sim_time = 0;
tick = 1;
alpha_1 = alpha_1_max;

%% Running the PSO
while((G.best_cost ~= 0) && (sim_time <= max_time))

    sim_time = sim_time + dt; %simulating passage of time (for change in
position)
    tick = tick + 1; %Counting number of dt seconds passed

    %Storing all the best cost to plot later
    global_cost(tick) = G.best_cost;

    % Linearlly decrease alpha 1 until a certain point each loop
    if (alpha_1 > alpha_1_min)
        alpha_1 = alpha_1 - alpha_1_delta;
    end

    % loop on each particle
    for i = 1:N

        % Retrieve particle name
        particle_name = particle_names(i);

        % Reset robot and wheel to initial conditions
        p.(particle_name).robot = InitialRobot;
        p.(particle_name).wheel = InitialWheel;

        % Ease of use
        pos = p.(particle_name).pos;
        robot =  p.(particle_name).robot;
```

```
        % COST
        total_error = 0;
        Wheel = p.(particle_name).wheel;

        % Simulation

        % Determine to draw or not (every 400 frames)
        if (~(mod(tick-2,200)) && (i==N))
            drawBool = 1;
        else
            drawBool = 0;
        end

        % Initialize the figure if drawBool
        if drawBool
            pause(1)
            figure()
            hold on
            title([particle_name, "iteration", tick-1])
        end

        % Animate the robot every 25 frames if drawBool
        for k = 1:2500
            if drawBool && ~mod(k-1, 25)
                pause(.001);
                drawRobot_Ackerman(robot, Wheel);
            end

            robot = fwdSim(robot, dt);
            [omega, gamma, error] = my_controller(robot, des, old_error, dt,
pos);
            total_error = total_error + abs(error);

            Wheel.gamma = gamma;
            robot.angVel = omega;
            old_error = error;

        end
        hold off

        % Calculating new cost
        new_cost = total_error/k;

        % Storing pos, robot, and wheel
        p.(particle_name).pos = pos;
        p.(particle_name).robot = robot;
        p.(particle_name).wheel = Wheel;

        % Updating PSO bests

        old_PB_cost = p.(particle_name).best_cost; %Old personal best cost

        %Storing cost if it's better than old PB
        if(abs(new_cost) < abs(old_PB_cost))
```

```
            p.(particle_name).best_cost = new_cost;
            p.(particle_name).best_pos = pos;
        end

        %Storing cost if it's better than old global best
        if(abs(new_cost) < abs(G.best_cost))
            G.best_cost = new_cost;
            G.best_pos = pos;
        end

        %Determing future velocity
        cur_vel = p.(particle_name).vel;
        cur_pos = pos;
        best_pos = p.(particle_name).best_pos;
        g_best_pos = G.best_pos;
        new_vel = alpha_1*cur_vel + alpha_2*rand(1)*(best_pos-cur_pos) +
alpha_3*rand(1)*(g_best_pos-cur_pos);
        new_pos = (new_vel - cur_vel) * dt + cur_pos;

        %Setting bounds on position and velocity
        if ((abs(new_vel(1)) < max_vel) && abs(new_vel(2)) < max_vel)
            p.(particle_name).vel = new_vel;
        end

        if ((new_pos(1)>=Params.xa  && new_pos(1)<=Params.xb) &&
(new_pos(2)>=Params.ya  && new_pos(2)<=Params.yb))
            p.(particle_name).pos = new_pos;
        end

        %Storing the positions into a Nx2xtick matrix
        positions(i,:, tick) = p.(particle_name).pos;

    end

    %Setting up a condition to stop looking if cost stops improving
    if (tick>5)
        if global_cost(tick-5) == global_cost(tick)
            global_improvement = 0;
        end
    end
end

fprintf("\n\nBest cost of %.2f found at (%.2f, %.2f)\n\n", G.best_cost,
G.best_pos(1), G.best_pos(2))


%% 1.3 Plotting/ Animating

% Plot parameters
plotBool = Params.plotBool;
plotResolution = Params.plotResolution;

if plotBool
    %Plotting particle posns (x,y) only
```

```
    pause(1);
    figure()
    hold on
    title("PSO Searching")
    xlabel('x')
    ylabel('y')
    grid on

    % Plot particle locations
    for z = 1:plotResolution:tick
        cla
        plot(positions(:, 1, z), positions(:, 2, z), 'x');
        hold on
        axis([(Params.xa*1.25 - 1.25) Params.xb*1.25 (Params.ya*1.25 - 1.25)
Params.yb*1.25])
        rectangle('position',[Params.xa, Params.ya, (Params.xb- Params.xa),
(Params.yb - Params.ya)])
        pause(.1);

    end
    hold off

    %Plotting the global cost vs time (ticks)
    max_cost = max(global_cost(2:end));
    min_cost = min(global_cost);
    pause(1)

    figure()
    hold on
    title("Cost v Iterations")
    xlabel("Iterations");
    ylabel("Global Cost")
    grid on

    % Animating global_cost v time
    for z = 1:plotResolution:tick-1
        cla
        plot(2:z, global_cost(2:z));
        axis([0 (z+2) (min_cost-2) (max_cost+2)]);
        pause(.5);


    end

    hold off
end

pause(1)
figure()
hold on
title("Best Gains Found")

robot = InitialRobot;
Wheel = InitialWheel;
```

```
% Plot the best combination of kp and kd regardless of plotBool
for k = 1:plotResolution:2500

    pause(.001);
    drawRobot_Ackerman(robot, Wheel);

    robot = fwdSim(robot, dt);
    [omega, gamma, error] = my_controller(robot, des, old_error, dt,
G.best_pos);
    total_error = total_error + abs(error);

    Wheel.gamma = gamma;
    robot.angVel = omega;
    old_error = error;

end

end



function [omega, gamma, error] = my_controller(robot, des, old_error, dt,
gains)

% K values
kp = gains(1);
kd = gains(2);
%ki = gains(3);

% Calculate error
error = des.Y - (robot.Y + (robot.width*sin(robot.Phi)));

% Don't allow Ki term to get too big
% if error_sum >= 100
%     error_sum = 0;
% else
%     error_sum = error_sum + error * dt;
% end

% Apply PID
gamma = kp*error + kd*(error-old_error)/dt; % + ki*error_sum;

% Introduce noise
% gamma = gamma + drift;

%gamma = atan2(sin(gamma), cos(gamma));

% Set bounds on steering angle
if (gamma>=pi/3)
    gamma = pi/3;
elseif (gamma<=-pi/3)
    gamma=-pi/3;
end
```

```
% Body angular rotation
omega = (robot.Vel/robot.width) * tan(gamma);


end



function [robot] = fwdSim(robot,dt)

%Current
X = robot.X;
Y = robot.Y;
Phi = robot.Phi;
Vel = robot.Vel;
angVel = robot.angVel;

%Future
X = X + Vel*cos(Phi)*dt;
Y = Y + Vel*sin(Phi)*dt;
Phi = Phi + angVel*dt;

%  if (Phi>=pi/2)
%       Phi = pi/2;
%   elseif (Phi<=-pi/2)
%           Phi=-pi/2;
%   end

%Passing
robot.X = X;
robot.Y = Y;
robot.Phi = Phi;


end



function [y_front_wheel] = drawRobot_Ackerman(robot, Wheel)


%% Main body
length = robot.length; %y-direction
width = robot.width; % x-direction

% Body vertices
y_box = [-length/2 -length/2 length/2 length/2 -length/2];
x_box = [width 0 0 width width];

% Robot initial conditions
x = robot.X;
y = robot.Y;
phi = robot.Phi;
```

```
% Rotating and translating robot body
rot_matrix = [cos(phi), -sin(phi); sin(phi), cos(phi)];
box_rotated = rot_matrix * [x_box; y_box];

box_translated_rotated = [box_rotated(1,:) + x; box_rotated(2,:) + y];


%% Wheels
radius = Wheel.radius;
wheel_width = Wheel.wheel_width; %y direction


%Back Wheel

x_back_wheel = [(-radius), ( -radius), (radius), (radius), (-radius)];
y_back_wheel = [wheel_width, -wheel_width, -wheel_width, wheel_width,
wheel_width];

back_wheel_rotated = rot_matrix * [x_back_wheel; y_back_wheel];

back_wheel_translated_rotated = [back_wheel_rotated(1,:) + x;
back_wheel_rotated(2,:) + y];


%Front Wheel
gamma = Wheel.gamma;

% Initial position
x_front_wheel = [(-radius), ( -radius), (radius), (radius), (-radius)];
y_front_wheel = [wheel_width, -wheel_width, -wheel_width, wheel_width,
wheel_width];

% Rotate wheel by steering angle
front_rot_matrix = [cos(gamma), -sin(gamma); sin(gamma), cos(gamma)];
front_wheel_steered = front_rot_matrix * [x_front_wheel; y_front_wheel];

% Put wheel at front of the body
front_wheel_translated = [front_wheel_steered(1,:) + width;
front_wheel_steered(2,:)];

% Rotate wheel with body
front_wheel_rotated = rot_matrix * front_wheel_translated;

% Combine rotation and translation
front_wheel_translated_rotated = [front_wheel_rotated(1,:) + x;
front_wheel_rotated(2,:) + y];


%% Plotting
cla
fill(box_translated_rotated(1,:), box_translated_rotated(2,:), 'b')
hold on
grid on
```

```
fill(back_wheel_translated_rotated(1,:), back_wheel_translated_rotated(2,:),
'k');
hold on
fill(front_wheel_translated_rotated(1,:),
front_wheel_translated_rotated(2,:), 'k')
axis([(-12+ x) (12+x) -12 12])
drawnow;
```

Published with MATLAB® R2024a