ROB 498 Final Project

Servando Garza (servando), Jose Galvan (jmgalvan), Edward Ivanac (eivanac)

***Abstract*—This Project is to implement the Kinematic Bicycle Model as a model for an Autonomous Vehicle. This 1st Checkpoint included our equations of motion for our model, the assumptions we've made so far, a linearized version of the model, and a discussion on the model's stability. We have included these sections in this checkpoint for completeness and context. For Checkpoint 2 we will outline the current state of our model, the waypoints our model traverses, the controllers we used to reach the waypoints, our tuning process for the controllers, and the performance of our vehicle when hitting the waypoints. For the final version of our model, we added an A* algorithmic planner to create a path that the model could use to traverse obstacles.**

I. Checkpoint 1

FOR the ROB 498 - Autonomous Vehicles semester-long project, our group has chosen to create a model of an automotive vehicle. To do so, we are replicating the pre-existing Kinematic Bicycle model, a common model in autonomous vehicle research. For this, we model a car as a set of two wheels, one front wheel that rotates to change the vehicle direction, and one back wheel that provides additional forward movement.

A. Equations of Motion

Our Kinematic Bicycle Model has the following equations of motion:

$\dot{x} = v * cos(\theta)$

$\dot{y} = v * cos(\theta)$

$\dot{\theta} = (v * tan(\delta))/L$

In these equations, we track three degrees of freedom: longitudinal roll (x), lateral pitch (y), and yaw (θ). We have two inputs into our model, steering wheel angle (δ) and speed (v); The parameter L is the length from the rear tire to the front tire.

B. Reference Frames and Assumptions

We take our reference frame with respect to the rear wheel of the vehicle. Because of the simplicity of the basic kinematic model, we made many assumptions when designing our model. In terms of model parameters, we assume that the radius of the turn, R, will be large compared to L, the wheelbase. We also assume that the left and right steering angles of the front wheels of the vehicle will be approximately the same, modeling them as one wheel. In terms of forces acting on the vehicle, our model as of Checkpoint 1 assumes that only the throttle force acts on the vehicle, propelling it forward. We assume there are no lateral forces or forces such as drag or friction acting on the system. We will discuss in the Conclusion/Next Steps section our plan to integrate these forces into the model.

C. Linearized Dynamics

We linearized our dynamics with small angle approximation around the point $\theta = 0$, $\delta = 0$. This results in the following equations of motion:

$x = x + v * t$

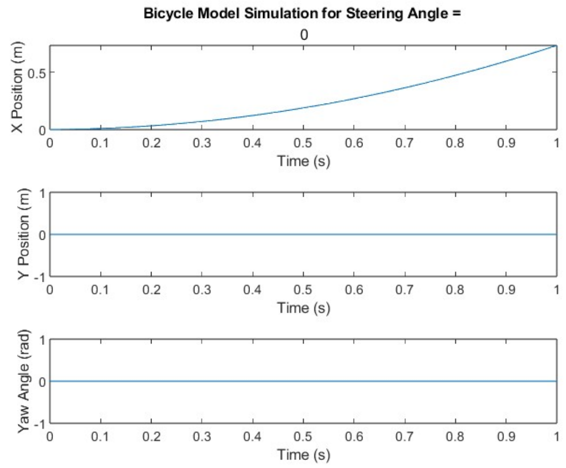
$y = y + v * \theta * t$

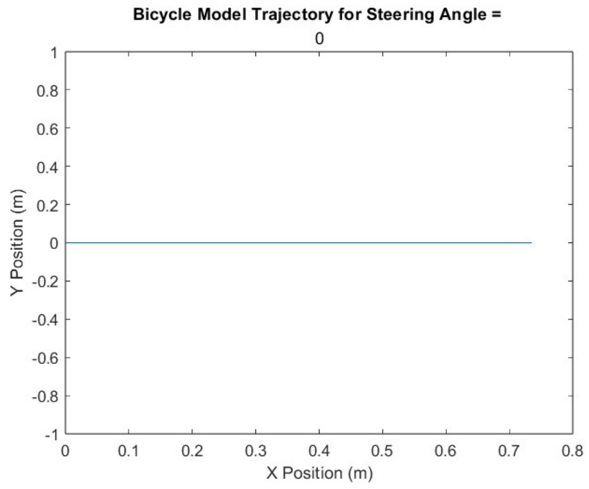
$\theta = \theta + ((v * \delta)/L) * t$

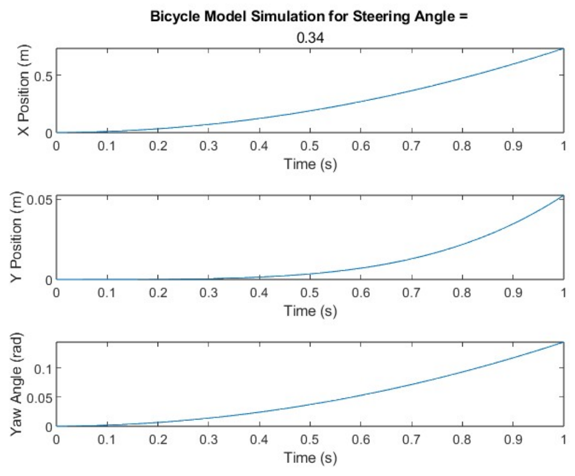
Code for simulating these linearized dynamics is provided in Appendix G.

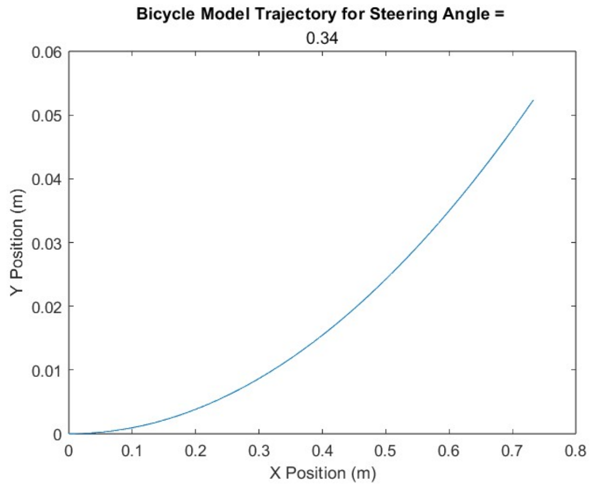
D. Simulated Open Loop Behavior

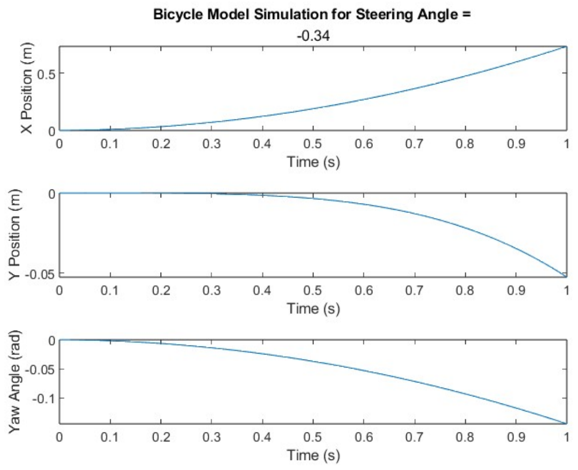
Using the code in Appendix F, we simulated our Kinematic Bicycle Model for a variety of different steering angles. All simulations were done for 1 second with a throttle force of 100 N. To reproduce our simulations, simply change the delta value to the desired steering angle.

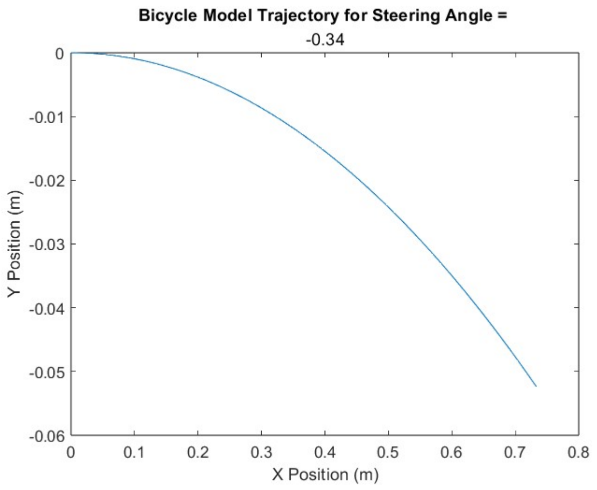












II. CHECKPOINT 2

A. Waypoint List and Conditions for Reaching Waypoint

The starting conditions of the vehicle are an X,Y coordinate of (0,0) and a yaw angle of 0. Our initial velocity is also zero. We selected 5 waypoints to test our controllers for the model. The list, in X,Y pairs is: (30,0), (41, 10), (35,20), (80,15), (130, -20). Our model traverses the list in this order, starting with (30,0) and ending at (130,-20). The condition for reaching these waypoints is that the center of our vehicle passes over the point. We verify this visually by plotting the trajectory of the vehicle as it traverses the waypoints. This can be seen in the Simulated Closed Loop Behavior section.

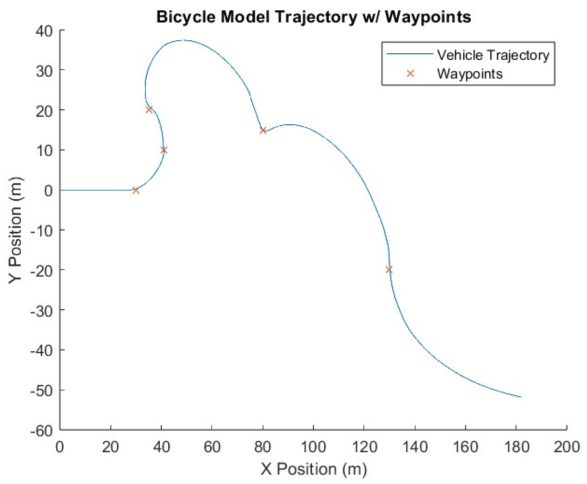
B. Controller Design

Our model has two controllers for navigating between waypoints. The first controller is a PI velocity controller that adjusts throttle force input into the model to reach a target velocity while moving between waypoints. The velocity controller uses a proportional gain and integral gain to adjust the throttle force acting on the model to reach the target velocity. The code for this velocity controller can be found in Appendix E. The second controller is a steering controller that adjusts the steering wheel angle input to reach a desired yaw angle. This controller is also a PI controller and similarly uses a proportional and integral gain the adjust the steering wheel angle to reach a target vehicle yaw angle. The code for this controller can be found in Appendix D.

C. Controller Tuning Process

In order to tune our controllers, we used a process of adjusting the PI gains in both controllers and rerunning the simulation until all waypoints were reached. We chose to start with the velocity controller and increase the P gain until we could reach the first waypoint. To reach the first waypoint, we needed to also increase the I gain to remove a steady state velocity error and be able to reach the waypoint. The first waypoint was chosen to test this controller because it didn't require the vehicle to turn, so we could tune the velocity controller independently of the steering controller. Once this waypoint was reached, we started to increase the P gain of the steering controller until we were able to hit the following waypoints. To remove any steady-state error in reaching all the waypoints, we increased the I gain in this controller. Once these controllers were tuned, we got the simulated behavior in the section below.

D. Simulated Closed Loop Behavior



In order to determine the distance and angle between waypoints we utilize the vehicle's current information and target. Based on this we are able to calculate the distance by computing the difference between the target and current position. For the angle between the two points, we utilize atan2 to compute the heading. Once the information has been gathered we pass the information into our velocity controller and steering controller functions respectively which provide a PI output. The controllers adjust the throttle force and steering angle of the car until all waypoints are reached. Once all waypoints are reached, the controllers adjust the steering wheel angle and throttle until the car hits a velocity and yaw angle of zero. This is what is seen in the photo above as the vehicle model navigates through the waypoint list. Code for this simulation can be found in Appendix A with reference functions in Appendix B, C, D, and E.

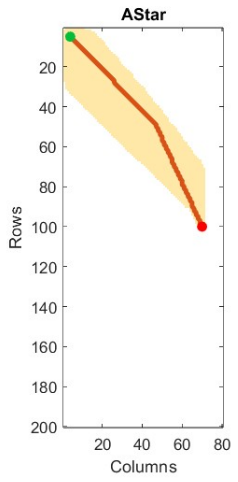
III. Final Model

For the final steps of our project, we decided to implement dynamic path planning into our kinematic bicycle model. To do so, we integrate the MATLAB built-in A* planning function into our model to plan paths around obstacles in a given map. With the path generated by the A* function, our model is able to traverse the path as a series of waypoints using our controllers and dynamics as discussed in the previous sections.

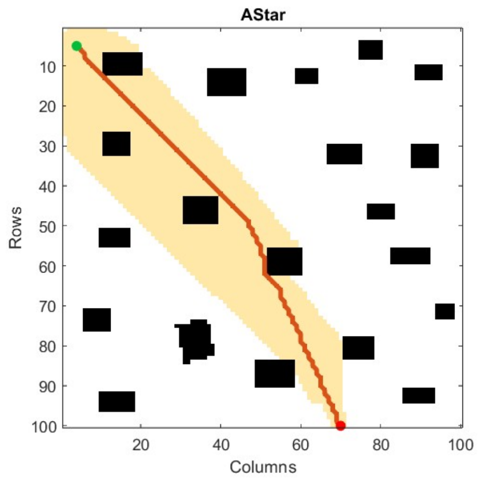
A. A* Path Planning

For this project we decided to use the MATLAB built-in A* function, plannerAStarGrid [1], to implement a path planner in our model. While implementing A* from scratch is a worthwhile exercise for any robotics engineer, all three of our members had previously implemented A* in our ROB 330 SLAM and Navigation Course at the University of Michigan, so our team felt comfortable enough using the built-in functions in this project. We create our maps to traverse and the plans to traverse said maps in the main file, as seen in Appendix A. To do so, we utilize four functions: binaryOccupancyMap() [2], mapClutter()[3], plannerAStarGrid(), and path()[4]. The function binaryOccupancyMap() is used to generate an 80m by 200m empty map. The function mapClutter() is used to create 20 obstacles randomly distributed obstacles throughout the map. The function plannerAStarGrid() is then used to create two sets of planners, one for the empty map and one for the obstacle map. We then define a starting X and Y coordinate position and a goal X and Y coordinate position. We then use the plan() function with inputs of our start position, goal position, and planner to plan two paths, one through the empty map and one through the obstacle map. Examples of these two maps and the paths planned for each of them are in the figures below.

Blank Map and Path



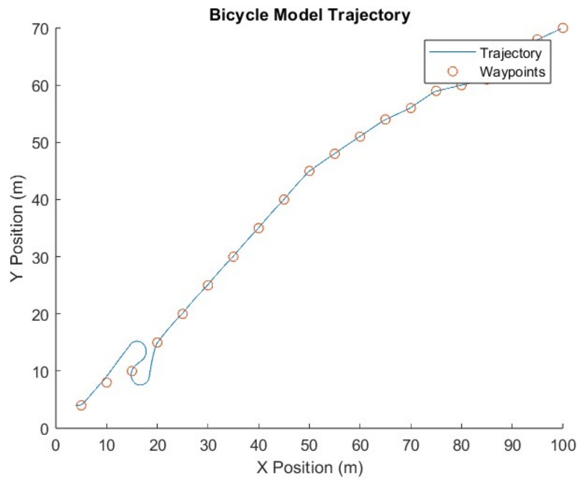
Obstacle Map and Path



B. Traversing the Path

After the paths to traverse the two types of maps are created, we must feed them into our model for traversing. To do so, these paths are saved into a mat file and loaded into a variable called pathway. We then downsample pathway to store every fifth waypoint into a list of waypoints that is loaded into model in full_sys() which can be found in Appendix

B. From the start of the simulation until one second in, the vehicle will be traversing waypoints from the path planned on the empty map. After one second in the simulation, the model switches to reading from the waypoint list generated in the map with obstacles. The model will react to the new waypoint list and navigate around the obstacles. The vehicle model traverses these waypoints using the steering controller and velocity controller found in Appendices D and E. One notable update is that the velocity controller is now a P controller and the steering controller is a PID controller. These changes were made as we were developing our path planning implementation using a similar tuning scheme as discussed in Section 2C: Controller Tuning Process. An example of our model following the the waypoints throughout the map can be seen in the image below.



IV. Conclusion

Throughout this project, our team had a main focus on learning more about the field of autonomous vehicles. As a team, we had relatively little experience with autonomous vehicles, so our team was excited at the chance to not only learn more about the field but apply it in this project. In this project, we learned more about the Kinematic Bicycle model, a common but simple model in autonomous vehicle research. We implemented friction forces such as drag into the dynamics of the kinematic bicycle for added complexity. We implemented a controller for our steering input and our velocity input, trying out various P, PI, and PID controllers with various gains until we found the correct controllers for our model. Finally, we integrated the A* planning algorithm, a common algorithm for planning paths around obstacles in a given map. This allowed us to neatly tie together the dynamics of the bicycle model and the controllers into an autonomous vehicle model that can plan a path, take in that path a series of waypoints, and autonomously traverse them. We'd like to thanks the instructional staff of the first ROB 498: Autonomous Vehicles class, Professor Anouck Girard and Nishant Kheterpal, for their support in making this project and class possible.

References

[1] "A* path planner for grid map - MATLAB," www.mathworks.com. https://www.mathworks.com/help/nav/ref/plannerastargrid.html#d126e149055 (accessed Dec. 11, 2023).

[2] "Create occupancy grid with binary values - MATLAB," www.mathworks.com. https://www.mathworks.com/help/nav/ref/binaryoccupancymap.html

[3] "Generate map with randomly scattered obstacles - MATLAB mapClutter," www.mathworks.com. https://www.mathworks.com/help/nav/ref/mapclutter.html (accessed Dec. 11, 2023).

[4] "Plan path between two states - MATLAB plan," www.mathworks.com. https://www.mathworks.com/help/nav/ref/plannerrrt.plan.html

Appendix A
Closed Loop Model Simulation Main File

```
clc;
clear ROB_498_checkpoint.m;
clear full_sys.m;
clear waypoints.m;
clear steering_control.m;

close all;

%Simulation with this model (nonlinear)

%Find a path
blank = binaryOccupancyMap(80,200); %
    Generates an empty map
obstacle = mapClutter(20, {'Box'},'
    MapSize',[20,20]); %Creates a map with
     obstacles using the mapClutter
    function
blankPlanner = plannerAStarGrid(blank); %
    Creates a planner to traverse the
    empty map
obstaclePlanner = plannerAStarGrid(
    obstacle); %Creates a planner to
    traverse the path with obstacles

start = [5,4];
goal = [100, 70];
blankWay = plan(blankPlanner,start,goal);
obstacleWay = plan(obstaclePlanner,start,
    goal);
save('pathfile.mat', 'blankWay', '
    obstacleWay');
show(blankPlanner)
figure
show(obstaclePlanner)

path = load("pathfile.mat",'obstacleWay')
    ;
pathway = path.obstacleWay;
sampling_factor = 5;
WaypointlistX = pathway(1:sampling_factor
    :length(pathway), 1);
WaypointlistY = pathway(1:sampling_factor
    :length(pathway), 2);

% State Variables
X = 4;              % X position (m)
Y = 4;              % Y position (m)
```

```
Theta = 0;      % Yaw angle (rad)
Delta = 0;
V = 8;
int_e_v = 0;
int_e_s = 0;

% Time Parameter
t_end = 60;     % Simulation time (s)
dt = 0.00001;

%Creating an event
odeFcn = @(t,y) full_sys(t, y);
% options = odeset('OutputFcn',@(t,y,flag
    ) deltaOutputFn(t,y,flag,WaypointlistX
    ,WaypointlistY));
[t, output_state] = ode45(odeFcn, 0:dt:
    t_end, [X;Y;Theta;V; int_e_v; int_e_s
    ]);

% Initialize arrays to store results
X_history = output_state(:,1);
Y_history = output_state(:,2);
Theta_history = output_state(:,3);
velocity_history = output_state(:,4);

% Plot results
figure;
subplot(4, 1, 1);
plot(t, X_history);
xlabel('Time (s)');
ylabel('X Position (m)');

subplot(4, 1, 2);
plot(t, Y_history);
xlabel('Time (s)');
ylabel('Y Position (m)');

subplot(4, 1, 3);
plot(t, Theta_history);
xlabel('Time (s)');
ylabel('Yaw Angle (rad)');

subplot(4, 1, 4);
plot(t, velocity_history);
xlabel('Time (s)');
ylabel('Velocity (m/s)');


figure
hold on
plot(X_history,Y_history)
title('Bicycle Model Trajectory');
xlabel('X Position (m)');
ylabel('Y Position (m)');
% WaypointlistX = pathway(1:10:end,1); %(
    m)
% WaypointlistY = pathway(1:10:end,2); %(
    m)
plot(WaypointlistX,WaypointlistY, "o")
legend("Trajectory","Waypoints")
hold off
```

Appendix B
Closed Loop Model Simulation Full System File

```
function dydt = full_sys(time, input)

persistent condition
persistent WaypointlistX
persistent WaypointlistY
persistent onStart

probObstacle = rand();

if isempty(onStart)
    onStart = 1;
    condition = 0;
    generic = load('pathfile.mat', '
       blankWay');
    points = waypoints(generic.blankWay,
       5, 0);
    WaypointlistX = points(:,1);
    WaypointlistY = points(:,2);

elseif onStart == 1 && probObstacle > .35
     && time > 1.0
    onStart = 2;
    disp(time);
    disp(probObstacle);
    generic = load('pathfile.mat', '
       obstacleWay');
    points = waypoints(generic.
       obstacleWay, 5, 0);
    WaypointlistX = points(:,1);
    WaypointlistY = points(:,2);
end

speed = 7;

% State Variables
X = input(1);          % X position (m)
Y = input(2);           % Y position (m)
Theta = input(3);       % Yaw angle (rad)
v = input(4);
err_throttle_int = input(5);
err_theta_int = input(6);

    %Stop car if all waypoints have been
       hit
    if (WaypointlistX ~= inf)
        xdiff = WaypointlistX - X;
        ydiff = WaypointlistY - Y;
        ang = atan2(ydiff,xdiff);

        error = ang-Theta;
        diff = sqrt((xdiff.^2)+(ydiff.^2)
            );
        if(diff <= .5)
            disp("Waypoint Reached");
            points = waypoints(NaN, NaN,
                1);
            WaypointlistX = points(:,1);
            WaypointlistY = points(:,2);
        end
        vel_ctrl = velocity_control(speed
            ,v,err_throttle_int);
        steer_ctrl = steering_control(
            error,err_theta_int);
        throttle = vel_ctrl(1);
        err_throttle = vel_ctrl(2);
        delta = steer_ctrl(1);
            if delta > 0.7854
                delta = 0.7854;
            elseif delta < -0.7854
                delta = -0.7854;
            end
        err_theta = steer_ctrl(2);
        out = bicycleDynamics([Theta;
            delta;v;throttle]);
    else
        disp("All Waypoints Reached!")
        err_throttle = 0;

        err_theta = 0;
        out = [0;0;0;.02];
        waypoints(NaN,NaN,2);
    end
    dydt = [out; err_throttle;err_theta];
end
```

APPENDIX C
CLOSED LOOP MODEL SIMULATION BICYCLE DYNAMICS FILE

```
function dydt = bicycleDynamics(in)

% Extract state variables from y
theta = in(1);
delta = in(2);
v = in(3);
throttle = in(4);

% Bicycle Model Parameters
m = 1519.988;   % Mass of the bicycle and
     rider (kg)
lf = 0.9;       % Distance from the
    center of mass to the front wheel (m)
lr = 0.9;       % Distance from the
    center of mass to the rear wheel (m)
rho = 1.2;      %density of air
Cd = .4;        %drag coeff for typical 4
     door car
A = 2.2914;     %cross sectional area of
    a Chevrolet Camaro
uw = 4.1667;    %wind velocity
g = 9.8;        %gravitational constant
W = m*g;        %gravitational force
f = 0.02;       %coeff of rolling
    resistance
theta_g = 0;    %Angle of vehicle with
    ground

Da = .5*rho*Cd*A*(v + uw)^2;    %
    Aerodynamic drag of vehicle

Xdot = v * cos(theta);
Ydot = v * sin(theta);
Omega = v * (tan(delta)/(lf+lr));
acceleration = (throttle - W*sin(theta_g)
     - f*W*cos(theta_g) - Da) / m;

% Create a column vector with derivatives
     of x, y, theta, delta, and v
dydt = [Xdot; Ydot; Omega; acceleration];

end
```

APPENDIX D
CLOSED LOOP MODEL SIMULATION STEERING CONTROLLER FILE

```
function steer_ctrl = steering_control(
    error,error_integral)

persistent errorlist

if isempty(errorlist)
    errorlist = 0;
end

% Gain

Dgain = .4;
Pgain = 175;
Igain = 37;


%Derivative
errorlist(end+1) = error;
der_error = errorlist(end) - errorlist(
    end-1);

% PID
P = Pgain*error;
I = Igain*error_integral;
D = Dgain*der_error;

% PID = P+I+D;
PI = P+I+D;
steer_ctrl = [PI, error];
```

```
end
```

APPENDIX E
CLOSED LOOP MODEL SIMULATION VELOCITY CONTROLLER FILE

```
function vel_ctrl = velocity_control(
    target_vel,current_velocity,
    error_integral)

% Gain
Pgain = 2000;


% Proportional
error = target_vel - current_velocity;


% PID
P = Pgain*error;


% PID = P+I+D;
vel_ctrl = [P, error];
end
```

APPENDIX F
KINEMATIC BICYCLE MODEL SIMULATION

```
clc;
clear;
close all;

% Bicycle Model Parameters
m = 70;          % Mass of the bicycle and
     rider (kg)
Iz = 1.2;        % Moment of inertia about
     the vertical axis (kg*m^2)
lf = 0.9;        % Distance from the
    center of mass to the front wheel (m)
lr = 0.9;        % Distance from the
    center of mass to the rear wheel (m)
Cf = 1600;       % Front tire cornering
    stiffness (N/rad)
Cr = 1800;       % Rear tire cornering
    stiffness (N/rad)
%Vx = 10;         % Forward velocity (m/s)
delta = .1;       % Steering angle (rad)
throttle_force = 100;  % Throttle force
    in Newtons

% State Variables
X = 0;           % X position (m)
Y = 0;           % Y position (m)
Psi = 0;         % Yaw angle (rad)
v = 0;

% Time Parameters
dt = 0.01;      % Time step (s)
t_end = 1;     % Simulation time (s)

% Initialize arrays to store results
time = 0:dt:t_end;
X_history = zeros(size(time));
Y_history = zeros(size(time));
Psi_history = zeros(size(time));

% Simulation loop
for i = 1:length(time)

    % Calculate acceleration using Newton
       's second law
    acceleration = throttle_force / m;

    % Integrate acceleration to calculate
        change in velocity
    delta_v = acceleration * dt;

    % Update velocity
    v = v + delta_v;

    %%Lateral forces to be factored in
       for Project Checkpoint 2
    % Compute lateral forces on the front
        and rear tires
    % Fyf = Cf * delta;
    % Fyr = -Cr * atan((lf * delta) / (lr
        + lf));
    %
    % % Compute lateral acceleration and
       angular velocity
    % Ay = (Fyf + Fyr) / m;
    % Omega = (lf * Fyf - lr * Fyr) / Iz;

    % Update state variables
    X = X + v * cos(Psi) * dt;
    Y = Y + v * sin(Psi) * dt;
    Psi = Psi + (v*tan(delta)/(lr+lf)) *
       dt;

    % Store results
    X_history(i) = X;
    Y_history(i) = Y;
    Psi_history(i) = Psi;
end

% Plot results
figure;
subplot(3, 1, 1);
plot(time, X_history);
xlabel('Time (s)');
ylabel('X Position (m)');
title('Bicycle Model Simulation');

subplot(3, 1, 2);
```

```
plot(time, Y_history);
xlabel('Time (s)');
ylabel('Y Position (m)');

subplot(3, 1, 3);
plot(time, Psi_history);
xlabel('Time (s)');
ylabel('Yaw Angle (rad)');

figure
plot(X_history,Y_history)
title('Bicycle Model Trajectory for
    Steering Angle = ',num2str(delta));
xlabel('X Position (m)');
ylabel('Y Position (m)');
```

APPENDIX G
LINEARIZED KINEMATIC BICYCLE MODEL SIMULATION

```
clc;
clear;
close all;

% Bicycle Model Parameters
m = 70;          % Mass of the bicycle and
     rider (kg)
Iz = 1.2;        % Moment of inertia about
     the vertical axis (kg*m^2)
lf = 0.9;        % Distance from the
    center of mass to the front wheel (m)
lr = 0.9;        % Distance from the
    center of mass to the rear wheel (m)
Cf = 1600;       % Front tire cornering
    stiffness (N/rad)
Cr = 1800;       % Rear tire cornering
    stiffness (N/rad)
Vx = 10;         % Forward velocity (m/s)
delta = .34;       % Steering angle (rad)
throttle_force = 100;  % Throttle force
    in Newtons

% State Variables
X = 0;           % X position (m)
Y = 0;           % Y position (m)
Psi = 0;         % Yaw angle (rad)
v = 0;

% Time Parameters
dt = 0.01;       % Time step (s)
t_end = 10;      % Simulation time (s)

% Initialize arrays to store results
time = 0:dt:t_end;
X_history = zeros(size(time));
Y_history = zeros(size(time));
Psi_history = zeros(size(time));

% Simulation loop
for i = 1:length(time)

    % Calculate acceleration using Newton
       's second law
    acceleration = throttle_force / m;

    % Integrate acceleration to calculate
        change in velocity
    delta_v = acceleration * dt;

    % Update velocity
    v = v + delta_v;

    %%Lateral forces to be factored in
       for Project Checkpoint 2
    % Compute lateral forces on the front
        and rear tires
    % Fyf = Cf * delta;
    % Fyr = -Cr * atan((lf * delta) / (lr
        + lf));
    %
    % % Compute lateral acceleration and
       angular velocity
    % Ay = (Fyf + Fyr) / m;
    % Omega = (lf * Fyf - lr * Fyr) / Iz;

    % Update state variables
    X = X + v * dt;
    Y = Y + v * Psi* dt;
    Psi = Psi + ((v*delta)/(lr+lf)) * dt;

    % Store results
    X_history(i) = X;
    Y_history(i) = Y;
    Psi_history(i) = Psi;
end

% Plot results
figure;
subplot(3, 1, 1);
plot(time, X_history);
xlabel('Time (s)');
ylabel('X Position (m)');
title('Bicycle Model Simulation');

subplot(3, 1, 2);
plot(time, Y_history);
xlabel('Time (s)');
ylabel('Y Position (m)');

subplot(3, 1, 3);
plot(time, Psi_history);
xlabel('Time (s)');
ylabel('Yaw Angle (rad)');

figure
plot(X_history,Y_history)
title('Bicycle Model XY Plot for Steering
    Angle = ',num2str(delta));
xlabel('X Position (m)');
ylabel('Y Position (m)');
```