1. Introduction:

The objective of this project is to design and implement a controller for an Ackermann kinematic model for the Acme Robotics company. This controller can possibly be a component of a self driving car software stack. It can take inputs such as a goal location and goal heading from a path planning module and apply the appropriate control input to the motors of the vehicle. The input for our Ackermann controller is a target heading angle and velocity. The output will be the steering angles and angular velocities for each of the front wheel. We assume that the ideal steering angle for a given set of inputs is unique.
Ackermann steering geometry is a geometric arrangement of linkages in the steering of a car or other vehicle designed to solve the problem of wheel slippage while executing a turn. The model has independent steerable wheels, to execute a turn, the inner and outer wheels have to trace out concentric circles of different radii. This ensures that the wheels do not slip.
The kinematic equations for this controller can be derived with basic trigonometry. These equations will be implemeted in C++ , because of its performance in real time systems. OOP practices will be used to make the code scalable.
The controller will use a PID control algorithm for calculating the ideal inputs to the system at every time step. The PID control has many advantages, the major advantage being its simplicity. But, the biggest disadvantage of PID is that we cannot add any constraints to the output signals and sometimes it can reach very high values, which may not be possible to attain in physical systems. Here one of our constraints is that maximum steering angle constraint < 45 degrees.

2. Development:

This development of this project will be in the form a pair programming process.

For Phase 0 -
Driver - Badrinarayanan Raghunathan Srikumar
Navigator - Smit Dumore

For Phase 1 -
Driver – Smit Dumore
Navigator - Badrinarayanan Raghunathan Srikumar

For Phase 2 -
Driver - Badrinarayanan Raghunathan Srikumar
Navigator - Smit Dumore

This project will be developed in a Linux environment and will make use of C++ 11/14/17 features. Tools like Travis, Coveralls , cppcheck, cpplint, GoogleTest and Github CI will be used to create a industry standard software.

3. Ackermann Model Geometry:

For a vehicle turning around a point with angle θ in time t, the radius of curvature is given by

$$R = \frac{v_t}{\omega} \qquad R = \frac{v_t * t}{\theta}$$

The indivisual steering angles can be calculated uisng the following formulaes,

$$\delta_i = \arctan\left(\frac{L}{R-(T/2)}\right)$$

$$\delta_0 = \arctan\left(\frac{L}{R+(T/2)}\right)$$

distance travelled by the car along the arc is given by

$$S_i = \left(R - {}^{T}\!/_{2}\right) * \theta$$

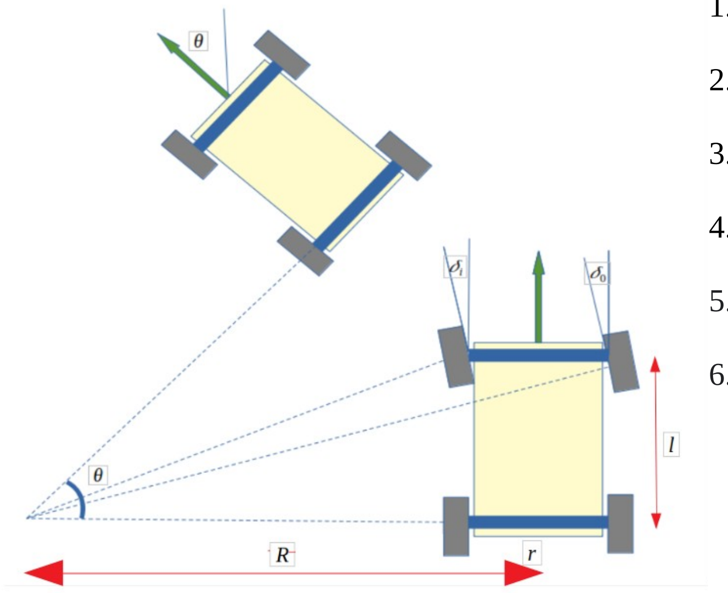
$$S_o = \left(R + {}^{T}\!/_{2}\right) * \theta$$

using the above results , the angluar velocity of indivisual wheels can be obtained as the following,

$$\omega_i = \frac{S_i}{r_w * t}$$

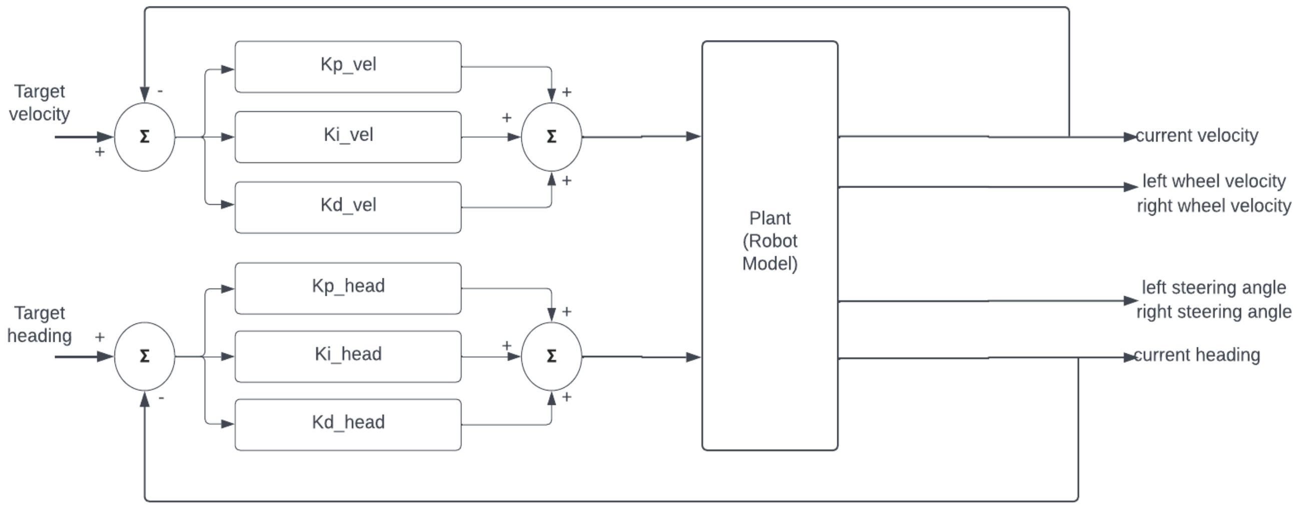
$$\omega_0 = \frac{S_0}{r_w * t}$$

Notations:



1. ***R*** **–** Turning radius of the vehicle
2. ***l*** – Wheel base
3. **r** – track width
4. **θ** – heading of the vehicle
5. δ**i** = inner wheel steering angle
6. δ**o** = Outer wheel steering angle

PID CONTROLLER:



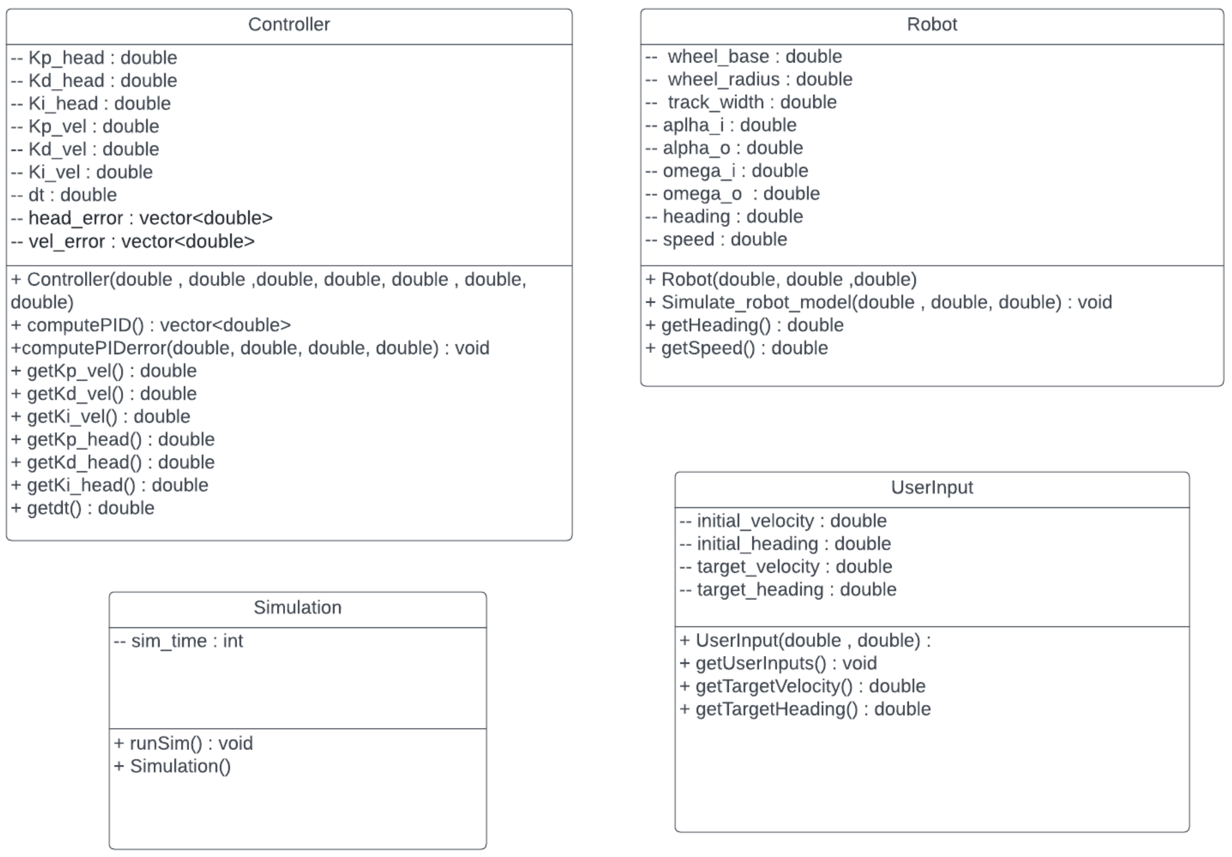
As shown in the diagram above, the PID controller takes in the error or difference between the set point and target set point. It uses the functionality of proportional, derivative and integral control together to reduce that error. The proportioal term (**P**) accounts for the present value of the error (i.e set point – process variable). The integral term (**I**) accounts for the past values of errors. Hence, the integral error is accumulated over time and the control output obtained from the integreal controller is such that it compensates for the accumulated error. The derivative term (**D**) takes into account the best estimate of the future errors by compensating for the rate of change of error occuring over time.

The figure above shows the output of the PID controller (u(t)), where **Kp** is the proportional constant, **e** is the error (set point – process variable), **Kd** is the derivative constant and **Ki** is the integral constant.

This logic is implemented in **Phase 2** of the project. The Plant (Robot Model) is simulated for a specific time interval to get a new robot state after every time step in the simulation. Using this updated robot we calculate the next control input. We demonstrate convergence to set point velocity and heading. This was acheieved by tuning controller gains.

CLASS STRUCTURE:

Significant changes have been made to the UML diagram, depicting the class structure.
Initially, we had only a Robot class and a Controller Class. The revised UML is as follows:



The **Robot** class defines the robot geometry, with its attributes being length of wheel base, radius of the wheel and track width. Its member functions being a constructor that can be used to define the initial values, and a computeTurnRadius function that is used to compute the turn radius.

The **Controller** class defines the charcteristics of the controllers, with its attributes being the proportional constant (Kp), derivative constant (Kd), integral constant (Ki) and the time interval. Its member functions include a constructor used to define the initial values of attributes and a compute function that computes the controller output. The getKp(), getKd(), getKi() and getdt() is used to return the respective attributes.

The **User Input** class is used to get inputs from the user.

The **Simulation** is used to run the controller for a specific amount of time, sim_time.

CONCLUSION :

Thus this demonstrates the feasibilty of this project and this can an integral part of the ACME robotics self driving car.