Vanderbilt University

School of Engineering

Technical Manual

Autonomous Car System

Grace DePietro

Yubo Du

Alex Feeley

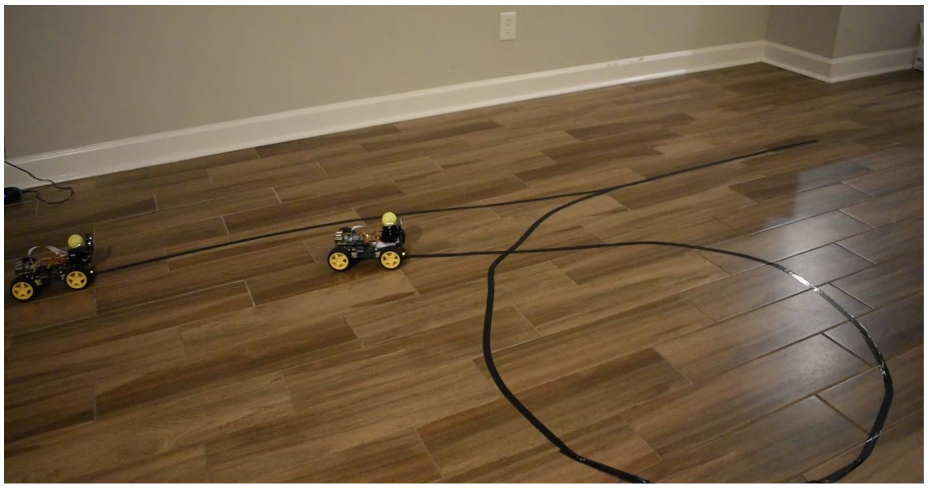
Yoni Xiong

December 11, 2020

Table of Contents

Principle of Operations

The Autonomous Car System tracks and follows a line and avoids collision with other cars. To do this, we utilize the IR sensor, Ultrasonic sensor, and Pi Camera module. Two cars begin at two separate tracks. One car travels around a loop, and both cars merge onto one track. A sample photo of our track is below:



The cars utilize a priority system to decide who yields to the other. This concept can be applied to civilian cars yielding to police cars or other emergency vehicles. In this scenario, the car traveling around the loop (low priority car), has a lower priority and must yield to the car traveling on the straight path (high priority car). The decision making algorithm is implemented for the low priority car and will allow it to decide what to do if it sees the high priority car at the intersection. It works by updating the car state parameters for the objects it sees in front of it in the CarState object. The two ways that are implemented to do this are with the ultrasonic class and the object detection and tracking. The ultrasonic distance gives the distance of any object in front of the car. The object detection and tracking uses the camera to track the yellow ping pong ball on the car and return the distance based on the size of the ball. After the CarState is updated with the various distance information, this is utilized by the decision algorithm to either slow down or completely stop the car. This algorithm is fully explained in the subsequent section.

Figure 1 shows the general process of the decision making process for the car system. The car system will continuously follow this decision process to traverse the system.

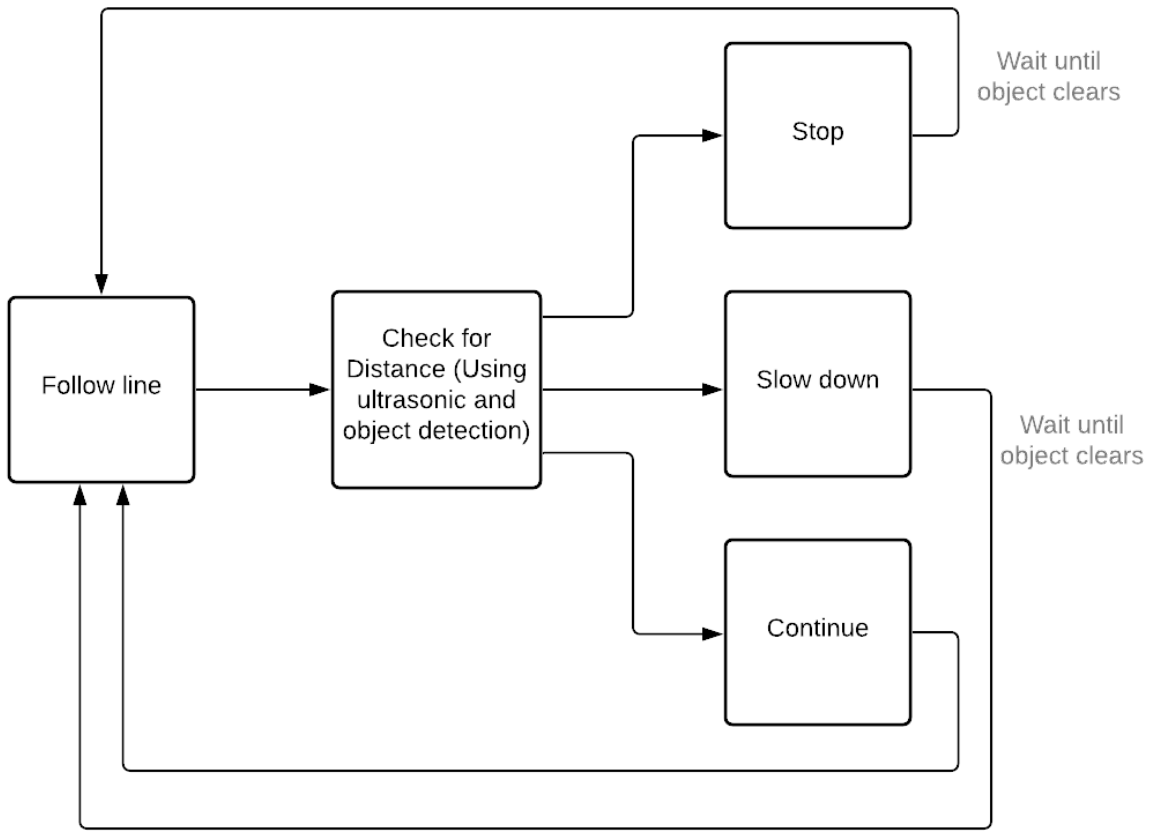


Figure 1: Decision Making Process Diagram

Decision Making Algorithm

The decision making algorithm was based on the priority of the car: high, medium, or low. Each decision making algorithm is represented by a state machine, as seen below. There were three states used in the decision making algorithm: Line Tracking where the car followed the normal line tracking algorithm, Slow Line Tracking where the car followed the slow line tracking algorithm, and Stop where the motors of the car have been stopped.

The high priority car continuously ran the line tracking algorithm without interruption. It did not make use of the camera or ultrasonic sensor since this vehicle was determined to have absolute priority on the road. This application is best suited for the use of ambulances or other emergency vehicles.

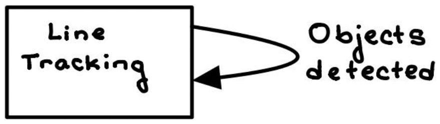


Figure 2: State Machine Diagram for High Priority

The medium priority car was to slow down if an object was directly in front of it, and stop if an object was detected to the side of it. The car used its ultrasonic sensor to determine if an object was directly in front of it and within 25 cm of the car. If this was the case, the car would slow down for five seconds or until the other object was more than 30 cm in front of the car, whichever was less. Additionally, the car used its camera and computer vision to determine if an object was detected not in front of the car and was within 25 cm of the car. The car would then stop until five seconds had elapsed or until the object was greater than 30 cm away, whichever was less. The state machine depicted in Figure 3 shows these transitions, and uses a clock invariant in order to ensure that a car will not stay in the slow or stopped states for longer than 5 seconds. This application was best suited for most civilian cars on main roads.

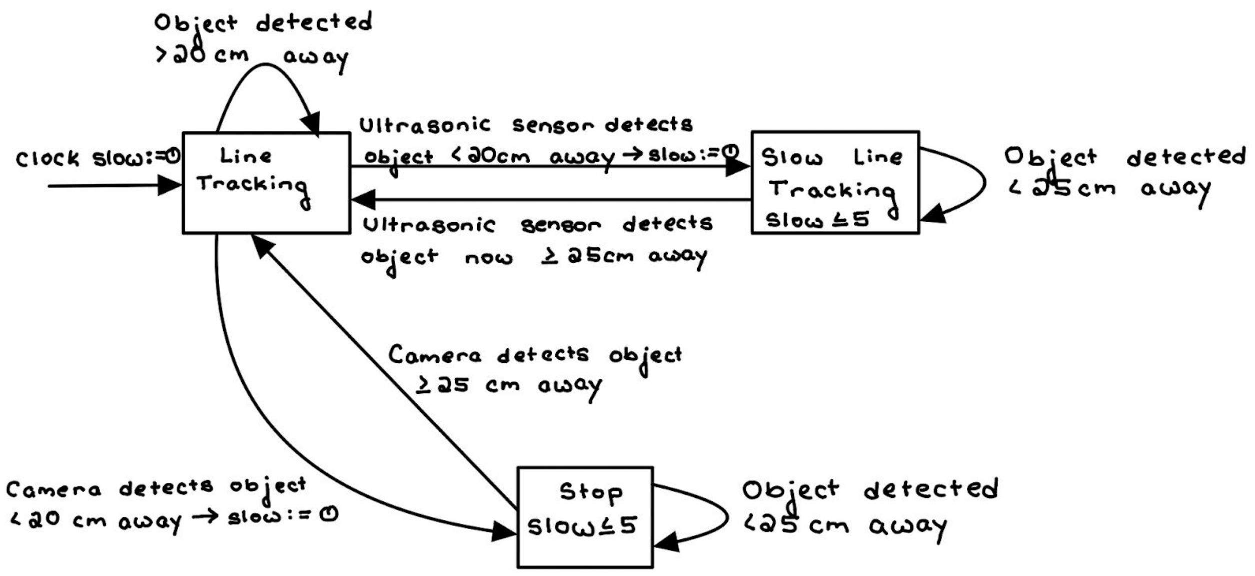

Object detected >20 cm away
clock slow:=0
Line Tracking
Ultrasonic sensor detects object <20cm away → slow:=0
Slow Line Tracking slow ≤ 5
Object detected <25cm away
Ultrasonic sensor detects object now ≥ 25cm away
Camera detects object ≥ 25 cm away
Camera detects object <20 cm away → slow:=0
Stop slow≤5
Object detected <25 cm away

Figure 3: State Machine Diagram for Medium Priority

The low priority car was to stop if a car came within 30 cm of it. The car detected objects using both its ultrasonic sensor and its camera. If an object came within 30 cm of the car, the car then remained stationary until the object was at least 40 cm away from the car or until 5 seconds had elapsed. As shown in Figure 4, the state machine for the low priority car has two states, and both the camera and ultrasonic sensor can cause the car to enter or exit the Stop state. A clock invariant is used in the Stop state to ensure that the car does not remain in this state forever.

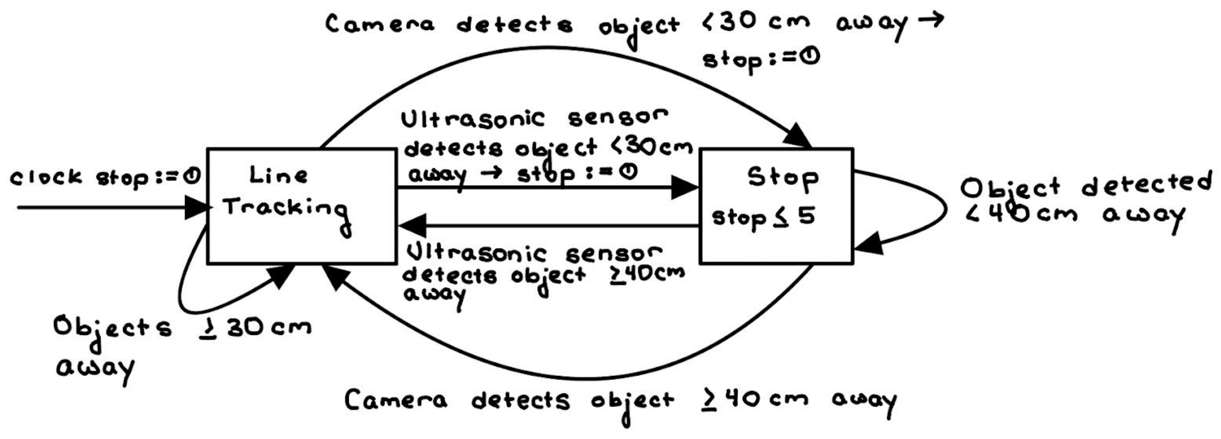

Camera detects object <30 cm away → stop:=0
Ultrasonic sensor detects object <30cm away → stop:=0
clock stop:=0
Line Tracking
Stop stop ≤ 5
Object detected <40cm away
Ultrasonic sensor detects object ≥40cm away
Objects ≥ 30 cm away
Camera detects object ≥ 40 cm away

Figure 4: State Machine Diagram for Low Priority

Line Tracking

The line tracking algorithm was taken from the open source repository:
https://github.com/Freenove/Freenove_4WD_Smart_Car_Kit_for_Raspberry_Pi

The line tracking algorithm remained unchanged except that a direction variable and method were added so that the direction of the car could be sent to the AWS server. The directions were as specified: 0 (Stop), 1 (Forward), 2 (Backward), 3 (Left), 4 (Right).

Additionally, a slow line tracking algorithm was created. This class utilized the Freenove Line tracking algorithm but altered the motor speeds so that the car would be slower. The slow line tracking algorithm also only has one turning speed whereas the regular line tracking algorithm has two. This allowed the car to travel much slower yet still allowed the motors to generate enough force to move the car. However, results may differ depending on the floor surface. This system used a hardwood floor as the testing course surface.

Network Communication

The network communication module includes the CarState class and network communication interface for the clients (Raspberry Pi cars) and the server hosted on an Amazon Web Services (AWS) instance.

CarState Class:
The CarState class manages an object called CarState that holds the current state of the car. The member variables of the object are listed and described in Table 1.

Table 1: CarState Class Details

Member Variable	**Description**
int car_id	Integer that represents the ID or priority of a car
int direction	Range 0-4, 0 = idle, 1 = forward, 2 = background, 3 = left, 4 = right
double speed	Value for speed of the car 0 = stop, 1 = slow, 2 = medium, 3 = fast
Ultrasonic ultrasonic	Reads and reports the distance of object sensed by ultrasonic sensor
int otherCarsCount	Number of other cars detected nearby using camera
List <double> carsNearbyLocation	List with location detected by the CV alg of a car nearby [x1,y1,r1,...]

Each member variable has a corresponding get() or set(...) function for reading or writing to the object. The primary functions of the class are send(address, port) and update (address,port, id), where the class can send the state of the car to the server or update an object with the state stored on the server. The class also features two logging functions to help with debugging. The CarState member variable values are printed out to the system console in a list separated by newlines using log() or a list separated by commas using logInLine(). The class does not have mutexes or synchronization because that can be done outside of the class if necessary.

The class has multiple helper functions to assist in sending and receiving data from the server using the CarState object. The class has a _str__() method to override the python string method, where the class parameters are serialized into a comma separated list for transfer to the server. The functions recv() and recvByID() call the Message classes recvMessage() function to poll the server for the state of the current car or by using a car_id parameter. The function updateState() updates the member variables of the current object using data encoded in a serial comma separated string from the server. The updateState() function is called by update (address, port) in order to update the current object after the car client receives the car state from the server.

Ultrasonic Sensor Class:
The Ultrasonic class controls the ultrasonic sensor on a car. The CarState class instantiates an instance of the Ultrasonic class in order to determine the current distance of an object in front of the car at all times. The Ultrasonic class works by sending out an ultrasonic wave forward. The wave hits the closest object in front of it, and bounces back towards the sensor. Then, the sensor “catches” the wave. We determine the time it leaves, time it returns, multiply it by the ultrasonic speed, and divide by two to get the final distance calculation in centimeters.

Server/Client Communication Protocol:
In order to establish communication between the car clients and the central server, a protocol was developed. The protocol uses a key and value pair with delimiters, inspired by the Uniform Resource Locator (URL) protocol. A series of commands is formatted like: key1=value1&key2=value2, where an ampersand is used to delaminate commands. Keys and values are separated by an equals sign. The key specifies the type of command. The value is the parameters for the command. Table 2 lists the possible key and values for the system.

Table 2: Communication Protocol Command Details

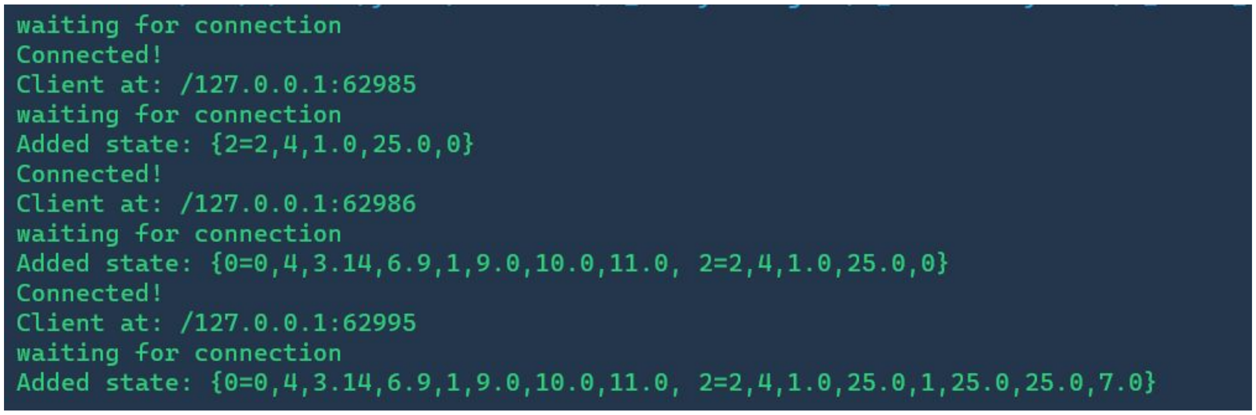
Key	Parameters
log_in	car_id
log_out	
send_state	serialCarStateString
send_ack	
request_state	car_id

A client may send a login command using the string: type=log_in&car_id=1. The server may send a car state to a client using the string: type=send_state&state=1,0,5,4,1,2,2,2.

Message Class:
The Message class uses the Python Socket library to perform low level data transmission with the server. Three functions exist in the class: sendMessage(address, port, car_id, serialMessage), recvMessage(address, port, car_id), and decodeMessage(serialMessage). The function sendMessage() works by logging into the server with the parameter car_id, formats a serial command to send the message with the serialized car state, sends the serial command to the specified port and IPv4 address, waits until the server sends an ack, and then sends a log out command. The revcMessage() works similarly, where it first logs in with the specified car_id parameter, sends a request command to the server, waits for a full message to be sent from the server, logs out, and returns the message sent by the server. The decodeMessage() function extracts and returns the serialized car state from the server for use in the CarState class.

Server Program:
The multithreaded server program uses many Java classes. Due to Java being out of the scope of this course, only the server functionality will be discussed in detail. First, to overview the other developed Java classes: the State class manages the same member variables as the CarState class in Python, the CarState class is a car identifier paired with a State object, and the MasterState class is a map that maps a car identifier integer to a CarState object. The server manages a shared MasterState object to maintain a record of all the car states of all cars that have updated to the server. The Message class encodes and decodes commands using the established server/client communication protocol.

The server functions by waiting for clients to connect, creating a ClientHandler thread instance to handle the connected clients, and repeating. The ClientHandler services all clients by responding to commands sent by the client until the client logs out. The ClientHandler will first expect a login command with the car identifier of the client. The ClientHandler can then use this identifier to process an update to that car in the MasterState map or retrieve the state from the MasterState map. Access to the shared MasterState map is managed using the synchronized() function to prevent writing or reading conflictions by concurrent threads. Finally, each ClientHandler will log the server actions to the console for viewing by the user. Figure 5 illustrates an example log report by the server. In the example, two clients are reporting to the server. The car with the identifier and priority of 2 detects one car nearby and updates the corresponding x, y, and r values at the end of the serial car state.



```
waiting for connection
Connected!
Client at: /127.0.0.1:62985
waiting for connection
Added state: {2=2,4,1.0,25.0,0}
Connected!
Client at: /127.0.0.1:62986
waiting for connection
Added state: {0=0,4,3.14,6.9,1,9.0,10.0,11.0, 2=2,4,1.0,25.0,0}
Connected!
Client at: /127.0.0.1:62995
waiting for connection
Added state: {0=0,4,3.14,6.9,1,9.0,10.0,11.0, 2=2,4,1.0,25.0,1,25.0,25.0,7.0}
```

Figure 5: Server Logging Example

Object Detection and Tracking

The object detection and tracking module is included in the BallCapture.py in ‘libraries’ folder.
Class name: BallCapture

1. Detection BallCapture.coloredBallTracking():
    a. Capture the current frame from the camera.
    b. Convert the image from RGB into HSV space.
    c. Select the yellow color area with a predefined parameter, set this area as a binary map: the yellow area as 1 and the other color area as 0.
    d. Erode and expand the image. There may be some small areas detected as the yellow color because of the light and the uncertainty of the camera. Eroding can reduce these noises in the binary map. After eroding the binary map the area of ‘real’ yellow color will decrease. Expand can restore the area of yellow color, while the noise areas are already eliminated they will not be restored.
    e. Use Canny Filter to select the edges in the binary map.
    f. Use the Hough Circle Transformation which detects all possible circles in an image, to track the circles in the binary map obtained from step e.
    g. Sort the selected areas and select the one with the largest radius as the colored ball.
    h. Save the current frame in the camera with the label of yellow colored ball.
    i. Return the x ,y axis location of the colored ball center in the image and the radius of the colored ball. If no ball detected the radius will be -1.

    The reason why color is detected in the binary map instead of using the method were the circle is detected in the grayscale image and the colored area is selected separately before combining the result together:

    1. Too much noise:
       There may be many noises in the grayscale image. Either filters or eroding can not deal with these noises dynamically. Binary map is very simple and is easy to eliminate the small noises.
    2. Time constraint:
       The noises will reduce the speed of the Hough Circle Detection algorithm. Additionally, performing two tasks sequentially increases the running time, and detecting circles in the binary map is very quick and efficient.

2. Depth Estimation BallCapture.dVision():

   Using the radius from the Detection task, the radius can be mapped the radius to the detected distance using:
   $-8.974553 - (2003.164/12.48963) * (1 - exp(-14.4896 * (1/radius))$.

   This function is obtained from many trials of testing: put the colored ball before the camera in 10 cm, 20 cm, 30 cm, 50 cm, 100 cm and record the radius at each distance. By fitting the distance and radius into the function $y = A - (B/C) * (1 - exp(-D * (1/x))$, the corresponding constants are derived.

   This project assumes that any other users who try to run this project will use the same smart car development kit, which means each car will use the same camera with the same angle and distance to the ground. As a result, the functions and parameters will not need to be calibrated or adjusted.

   If no ball is detected, this function will return an approximately infinite value.

3. Tracking:
   If the distance detected remains less than 30 cm, the function will notify the decision process. Otherwise, this module will respond that there is nothing near the car.

Users can either call Ballcapture.catureOne() which will finish the ball capture and distance computing part together and return [x,y,radius,distance], or call Ballcapture.coloredBallTracking() and Ballcapture.dVision(r) separately and sequentially.