Line Following Controller Design for Remote-Controlled Car

Ezequiel Juarez Garcia and Arman Sargolzaei, *Member, IEEE*

***Abstract*—This paper describes the process used to design a line following controller for a remote-controlled car. The car uses the Open Source Computer Vision Library in conjunction with a stereo camera to detect a colored line. Using the Robot Operating System and a Python program, the embedded system on board the car is able to send steering instructions to the DC servo motor that controls the front wheels. A proportional gain controller is used to minimize the error distance and keep the car aligned with the line.**

***Index Terms*—Line following controller, remote-controlled car, Open Source Computer Vision Library, Robot Operating System, Python program, DC servo motor, proportional gain controller.**

I. Introduction

A. Motivation and Goals

THE motivation for this paper started with a project proposal for my Control System Design class at Florida Polytechnic University. During the same semester, I was also taking an Autonomous Vehicles class as independent studies. Due to this, I wanted to do work on a project that would benefit me in both classes. After much thought, I chose to design a line following controller for the remote-controlled car I was using in Autonomous Vehicles.

My first goal for the project was to learn more about the autonomous vehicles. Although the scope of the project would involve little to no vehicle autonomy, I wanted to get familiar with the industry. By working on the car and through research, I would learn how a scale model of an autonomous vehicle is built and what tools are used to program the vehicles.

The second goal of the project was to help out my team members in the Autonomous Vehicles. Due to my lack of knowledge of the Robot Operating System (ROS) and the Open Source Computer Vision Library (OpenCV) before commencing the project, I was not able to provide much aid to my computer engineering team members. When we were assigned a lab that dealt with line following and PID controllers, I saw an opportunity to contribute to the project.

The third and final goal for the project was to learn more about ROS and OpenCV. While working on the project, I learned that they are both powerful tools used in educational and industrial settings. Learning to write ROS nodes, or programs, took some time, but the tutorials on the ROS wiki were instrumental to the project. The application programming interface (API) tutorials on OpenCV were also very helpful. With all of these resources and the help of the community, I was able to create the line following controller.

B. Literature Review

This project was made possible with the work done by other people. This section highlights the work done in similar fields and how it influenced the project and this paper.

Professors at Near East University in Turkey used a educational robot to implement a line following fuzzy controller [1]. The advantage of this controller is that is that does not require a model of the system. The use of a fuzzy controller was discussed for this project, but because the lab in Autonomous Vehicles required a PID controller, the idea was scrapped. A good extension to this project would be to implement a fuzzy controller. Its performance could then be compared to that of the P controller.

Another interesting approach to line following comes from an article by Robert Bosch Engineering and Business Solutions in India [2]. The articles discusses how a neural network can be used to create a robot that can follow curves smoothly. Although a section comparing the performance of PID controllers versus neural networks would have added more substance to this paper, it was, unfortunately, outside the scope of this project.

A paper written at SASTRA University [3] used computer vision software to assist a line following robot. This results from this paper helped finalize the decision of using a camera instead of a light detection and ranging (LIDAR) sensor for the line following controller.

C. Physical Components

The practical demonstration of the project used various physical components. The vehicle we used for the project was the rally-type, remote-controlled car manufactured by Traxxas. The main component in the car that we looked at was the DC servo motor that controls the front-wheel steering. In addition to the components that made up the car, it was also outfitted with a light detection and ranging (LIDAR) sensor, 3D depth camera, stereo camera, embedded system, inertial measurement unit (IMU), USB hub, and battery pack.

The main components used in this project were the battery pack, embedded system, USB hub, stereo camera, and the car. The stereo camera was the ZED camera made by Stereolabs. The camera was connected to the USB hub. The USB hub connected the camera and other sensors to the NVIDIA Jetson TX1 embedded system board. Even though size of the board was small, it proved powerful enough to run the line following controller.

The last physical component vital to the project was the colored line that the car tracked and followed. We used blue

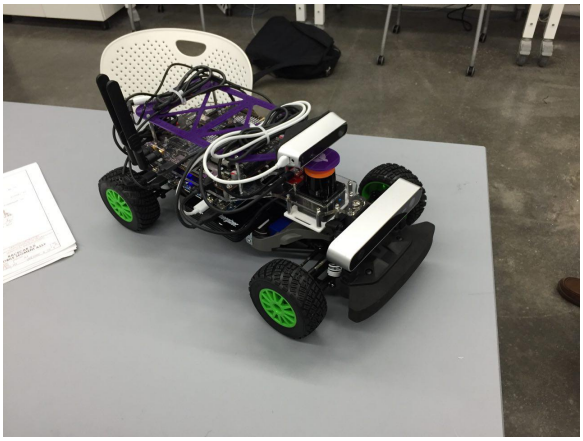

Fig. 1. Remote-controlled car used in the project. The shell was detached and sensors and an embedded system were placed on it. The ZED camera is mounted on the front of the car.

painter's tape commonly found at a local hardware store. The line on the floor consisted of straightaways, curves, and sharp corners.

D. Software Components

Alongside the physical components, the software components made up the second half of this project. The NVIDIA board was flashed with Linux, more specifically, the Ubuntu 16.04 distribution. The Ubuntu image was downloaded from an MIT website because it came preconfigured and loaded with Robot Operating System and other tools. ROS is simply a framework of tools used to develop software for robots [4]. With ROS, we can write code is several different programming languages and allow them to talk to one with one another using a common protocol. Programs written in ROS are referred to as nodes. The final software component was the P controller written in Python that is used to keep the car following the line on the floor. The design of the controller is covered under greater detail in the Controller Design section.

II. System Model

Initially, I wanted to model the entire system of my project. This would have included the car and the steering and line following controllers. However, this proved to be more difficult than I had anticipated and beyond my level of expertise with MATLAB and Simulink. With Dr. Arman's guidance and help, I focused solely on modeling the servo for the theoretical section of the project. The reason for only modeling the servo was that it would help me understand one the most important hardware components in my project and allow me to dedicate more time to the practical part of the project.

We will begin by analyzing the electrical and mechanical representation of a DC servo motor. After that, we will derive the state equations. A document from George Madison University [5] facilitated the derivations of the state equations.

A. Electrical and Mechanical Representation Explanation

The electrical and mechanical representation of a DC servo motor is shown in Fig. 2. The resistance and inductance of the armature winding are represented by R_a and L_a respectively. When a voltage $V_a(t)$ is applied to the armature, current $i_a(t)$ flows though the armature and drives the motor's load M and inertia J. B is the damping in the motor [5].

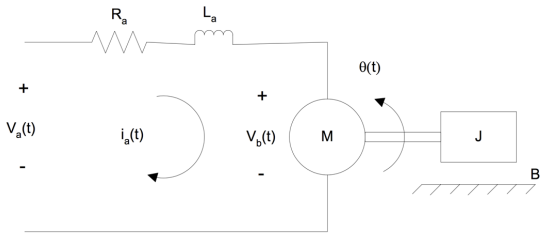

Fig. 2. Electrical and mechanical representation of DC servo motor [5].

B. State-Space Representation

We will begin by looking at the electrical system in Fig. 2 and performing a mesh analysis .

$$V_a(t) = R_a i_a(t) + L_a \frac{di_a(t)}{dt} + V_b(t), \tag{1}$$

where $V_b(t) = K_b \frac{d\theta(t)}{dt}$ and K_b is the motor's back electromotive force (EMF) constant. The substituted equation looks like the following:

$$V_a(t) = R_a i_a(t) + L_a \frac{di_a(t)}{dt} + K_b \frac{d\theta(t)}{dt}. \tag{2}$$

Now let's look at the mechanical system of Fig. 2.

$$J\frac{d^2\theta(t)}{dt^2} + B\frac{d\theta(t)}{dt} = T_{app}, \tag{3}$$

where $T_{app} = K_T i_a(t)$ and K_T is the torque constant. The new equation looks like the following:

$$J\frac{d^2\theta(t)}{dt^2} + B\frac{d\theta(t)}{dt} = K_T i_a(t). \tag{4}$$

We will rearrange (2) for it to be later represented in state-space form.

$$L_a \frac{di_a(t)}{dt} = -R_a i_a(t) - K_b \frac{d\theta(t)}{dt} + V_a(t) \tag{5}$$

$$\frac{di_a(t)}{dt} = -\left(\frac{R_a}{L_a}\right) i_a(t) - \left(\frac{K_b}{L_a}\right) \frac{d\theta(t)}{dt} + \left(\frac{1}{La}\right) V_a(t) \tag{6}$$

We will do the same for (4).

$$J\frac{d^2\theta(t)}{dt^2} = -B\frac{d\theta(t)}{dt} + K_T i_a(t) \tag{7}$$

$$\frac{d^2\theta(t)}{dt^2} = -\left(\frac{B}{J}\right) \frac{d\theta(t)}{dt} + \left(\frac{K_T}{J}\right) i_a(t) \tag{8}$$

Let's now define the state variables for the state equations as $x_1 = i_a$, $x_2 = \theta$, and $x_3 = d\theta/dt$, $u(t) = V_a(t)$, and $y(t) = \theta(t)$.

$$\dot{x_1} = -\left(\frac{R_a}{L_a}\right) x_1 - \left(\frac{K_b}{L_a}\right) x_3 + \left(\frac{1}{La}\right) u(t) \tag{9}$$

$$\dot{x_2} = x_3 \tag{10}$$

$$\dot{x_3} = -\left(\frac{B}{J}\right) x_3 + \left(\frac{K_T}{J}\right) x_1 \tag{11}$$

$$y(t) = x_2 \tag{12}$$

The general form of the state-space representation is the following:

$$\dot{x} = Ax + Bu \tag{13}$$

$$y = Cx + Du. \tag{14}$$

In this equation, x is the state vector, $\dot{x}$ is the derivative of the state vector with respect to time, y is the output vector, u is the input or control vector, A is the system matrix, B is the input matrix, C is the output matrix, and D is the feedforward matrix [6]. In our case, there is no feedforward matrix.

The resulting state-space representation for the servo is the following:

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \end{bmatrix} = \begin{bmatrix} -\frac{R_a}{L_a} & 0 & -\frac{K_b}{L_a} \\ 0 & 0 & 1 \\ \frac{K_T}{J} & 0 & -\frac{B}{J} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} \frac{1}{L_a} \\ 0 \\ 0 \end{bmatrix} u(t) \tag{15}$$

$$y(t) = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}. \tag{16}$$

C. Simulink Servo Model

The Laplace transform of (6) and (8) yield the following equations:

$$sI(s) = -\left(\frac{R_a}{L_a}\right) I(s) - \left(\frac{K_b}{L_a}\right) s\theta(s) + \left(\frac{1}{La}\right) V(s) \tag{17}$$

$$s^2\theta(s) = -\left(\frac{B}{J}\right) s\theta(s) + \left(\frac{K_T}{J}\right) I(s). \tag{18}$$

These equations can be used to model the servo as a subsystem in Simulink as seen in Fig. 3.

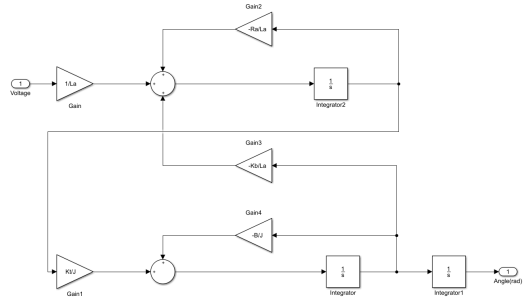

Fig. 3. Simulink model of DC servo motor.

III. Stability Analysis

A. MATLAB Code

The stability of the servo model was obtained using MATLAB and Simulink. Using an equation on page 139 in the book Control Systems Engineering [6] and a few MATLAB commands, the general formula for the servo transfer function was derived. The MATLAB code for the formula is shown in Fig. 4.

```
%% State-space model to transfer function using symbols

syms Ra La Kb Kt B J s;
A = sym([-Ra/La 0 -Kb/La; 0 0 1; Kt/J 0 -B/J]);
E = sym([1/La; 0; 0]);
C = [0 1 0];
D = 0;

tfsym = sym(C*((s*eye(3)-A)^(-1))*E+D);
```

Fig. 4. MATLAB code used to derive the servo transfer function from the state-space representation.

The following equation is the transfer function of a DC servo motor.

$$G(s) = \frac{\frac{K_t}{JL_a}}{s^3 + \left(\frac{B}{J} + \frac{R_a}{L_a}\right) s^2 + \frac{BR_a + K_bK_t}{JL_a} s} \tag{19}$$

The values for the parameters of the actual servo were not found. This was due to the lack of a proper datasheet from the manufacturer. Instead, to check the stability of the servo transfer function $G(s)$, example parameter values were used that were not far off from realistic values.

```
%% Transfer function verification

La = 0.5;
Ra = 1;
Kb = 0.01;
Kt = 0.01;
B = 0.1;
J = 0.001;

A = [-Ra/La 0 -Kb/La; 0 0 1; Kt/J 0 -B/J];
E = [1/La; 0; 0];
C = [0 1 0];
D = 0;

sys = ss(A,E,C,D);
tfsys = tf(sys);

%% Stability Check

pzmap(tfsys);
```

Fig. 5. MATLAB code used to check the stability of the system without a controller.

The MATLAB code shown in Fig. 5 uses example servo parameters to compute the state-space representation. It then plots the pole-zero map of the transfer function. The pole-zero map is shown in Fig. 6. Note that the servo model is unstable because not all of the poles lie strictly on the left-hand plane.

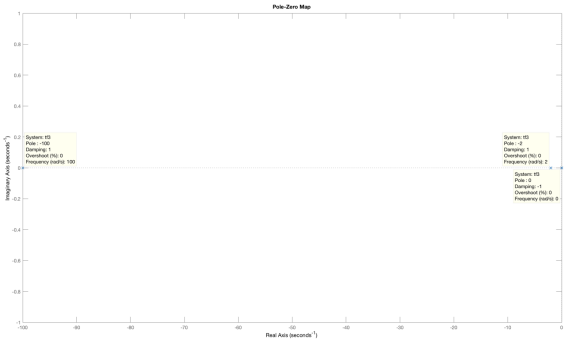

Fig. 6. Pole-zero map of servo transfer function using example values.

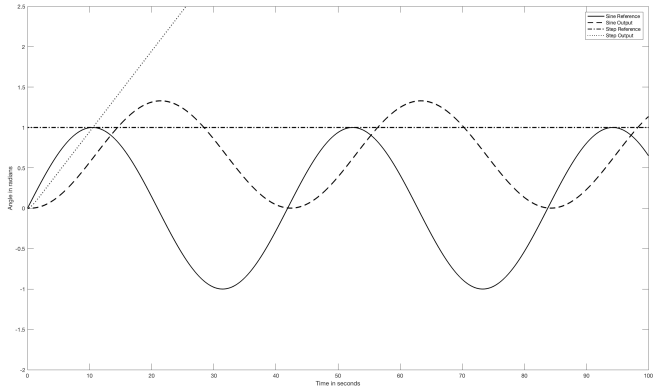

Fig. 8. Output plot of the step and sine inputs to the servo.

B. Simulink Model

The Simulink model of the servo confirmed the stability findings from the MATLAB code section. This model shown inFig. 7, includes two input signals. The first input to the system is a sine input. This type of input is the most practical one because it mimics the type of steering when driving on roads with curves. The second input is a step input. This input is typically used when calculating the maximum angle that a servo can turn and consequently steer the front wheels.

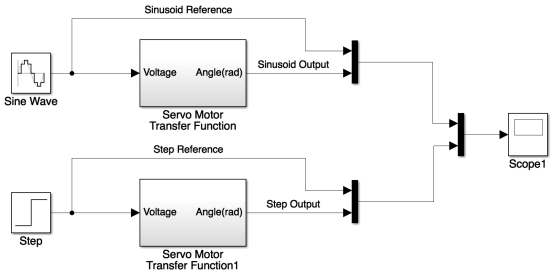

Fig. 7. Simulink servo model including inputs and outputs.

Fig. 8 shows the input and output signals of the Simulink model. Note that the servo tries to track the sine input but it is always lagging. This behavior is typical of servo motors because they cannot instantaneously turn to the desired angle. The servo used in the project car stated a turning angle of 0.34 radians in 0.17 seconds. Those numbers are different than the ones shown on the plot.

The step output shows instability in the system. This is due to the fact that normal servos have a physical limiter that prevents them from spinning forever when a step voltage is applied. A saturation block can be added to the Simulink model keep the output between a certain range of values.

The instability of a servo when given a step input brings up the argument that instability in a system can sometimes be beneficial. Just like an oscillator, an unstable system, can be used to create clock signals for computers, an unstable servo can be used to drive a load just like a regular DC motor. All things considered, a servo is basically a motor; therefore, it behaves similar.

IV. Controller Design

The design of the line following controller took place in software. From the beginning, we knew that we had to design two ROS nodes. One would track the line on the floor with the ZED camera and the other would send steering instructions to the electronic speed control unit. With the help of senior student Michael Sanchez, the design of the controller was made possible.

In the beginning of the line following node, we read in a frame from the camera and crop it. Cropping scales down the image to decrease computational intensity and keep only the line within the frame view. After cropping the image, the program performs a Gaussian blur to get rid of noise in the frame. It then converts the frame from the blue-green-red (BGR) colorspace to the hue-saturation-value (HSV) colorspace. In the HSV colorspace, it is much easier to detect an object by color because we can set the first number, the hue, to the color of the line and set the saturation and value to a wide range of values to account for different lighting conditions.

After being converted to HSV, the image undergoes several other processes to find the contour of the blue line. Fig. 9 shows a code snippet of the line following node. The code snippet picks up right after the BGR to HSV conversion and calculates the center point of the contour. This is where cropping is important. The more we crop the top of the image, the closer the center of the contour will be to the car. We chose to keep the center of the line at approximately 1.5 feet from the front of the car.

The distance error from the line is actually a pixel count. We measure the amount of pixels that the center of the contour is from the middle of the frame. If the center of the contour is in the right-hand plane of the frame, that means the car is veering off to the left and the program publishes a negative pixel error. A positive pixel error is published when the car veers off to the right.

The steering node is in charge of taking the pixel error from the line following node and steer the car in the direction of the line by turning the wheels. The rotation angle of the servo is controlled by sending command to the electronic speed control unit. The most important piece of code in the line

```
## Only proceed if at least one contour was found
if len(cnts) > 0:

    ## Find the largest contour in the mask, then compute the centroid
    c = max(cnts, key=cv2.contourArea)
    ((x, y), radius) = cv2.minEnclosingCircle(c)
    M = cv2.moments(c)
    center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))

    ## Calculate the pixel error, publish the error, and sleep
    error_int = center[0] - (frame_width/2)
    # rospy.loginfo(error_int) # print error while running node
    pub.publish(error_int)
    rate.sleep()
```

Fig. 9. Code snippet of the line following node. Various transformations are performed image/frame read from the camera in order to calculate the middle of the line and pixel error.

```
def callback(data):

    ## P-controller design. The proportational gain is basically
    ## (max steering angle/pixel value less than half of the frame width).
    ## The pixel value has been tested with numbers up to 290. Lower numbers
    ## gives turning around sharp corners.
    msg.drive.steering_angle = (0.34/150)*(-data.data)

    ## Public topic message
    pub.publish(msg)
```

Fig. 10. Code snippet of the steering node. It shows the code for the P controller.

following node is shown in Fig. 9. In the snippet we can see the P controller code that proved more than capable for the purpose of line following. The proportional gain is calculated by dividing the maximum turning angle of the servo in either direction into half the size of the frame width or less.

The intuition behind the proportional gain of the controller is that the car will turn up to 0.34 radians in either direction depending on where the center of the line is located within the frame. A high gain gives us a faster response time but can lead to more oscillations. A small gain will make the car lose sight of the line more frequently. In all the test drives, the speed of the car remained constant.

V. Conclusion

This paper shows the design of a line following program for a remote-controlled car. A proportional gain controller is used to minimize the error between the car and the line. A P controller was chosen due to its simplicity. It was implemented in software using open source components like the Robot Operating System and Open Source Computer Vision Library. Using what we learned in the project, we can apply an adaptive PID controller to an RC car to make it more autonomous.

References

[1] D. Ibrahim and T. Alshanableh, "An undergraduate fuzzy logic control lab using a line following robot," *Computer Applications in Engineering Education*, vol. 19(4), pp. 639–646, 2011.

[2] A. Roy and M. M. Noel, "Design of a high-speed line following robot that smoothly follows tight curves," *Computers & Electrical Engineering*, vol. 56, pp. 732 – 747, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045790615002190

[3] M. S. Prakash, K. A. Vignesh, J. Shyamsunthar, K. Raman, J. S. Raju, and N. Raju, "Computer vision assisted line following robot," *Procedia Engineering*, vol. 38, pp. 1764 – 1772, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877705812021285

[4] About ros. [Online]. Available: http://www.ros.org/about-ros/

[5] Deriving state equations for a dc servo motor. [Online]. Available: http://ece.gmu.edu/ gbeale/ece_521/xmpl-521-dc-motor-model-01.pdf

[6] N. S. Nise, *Control Systems Engineering*. Hoboken, NJ: John Wiley & Sons, Inc., 2011.