A Control Pipeline for Robust Lane Keeping in Model Cars

James Vero Asghar, Paul Auerbach, Maximilian Matthé
Connected Robotics Lab
Barkhausen Institut gGmbH
Dresden, Germany
firstname.lastname@barkhauseninstitut.org

Carsten Knoll
Chair of Fundamentals of Electrical Engineering
Technical University Dresden,
Dresden, Germany
carsten.knoll@tu-dresden.de

***Abstract*—Reliable lane keeping is a fundamental requirement of autonomous model cars. This paper presents a sophisticated image processing pipeline in combination with the Stanley steering controller to reliably steer model cars on their lane at high speeds. We show that the pipeline significantly improves robustness compared to a naive steering implementation. The results ensure reliable lane keeping and hence enable testing of higher-level algorithms for e.g. overtaking or merging scenarios.**

***Index Terms*—lane keeping, stanley controller, model vehicles, image pipeline**

I. Introduction

With the emergence of autonomous and automated vehicles, the research on algorithms for improving reliability, efficiency and safety has greatly increased. Despite having accurate traffic and vehicle models used for simulation, results need to be tested and verified in reality [1]. However, nowadays cars are very complex, offer strict software interfaces, and need to conform to safety regulations [2], which makes them unsuitable for initial testing of algorithms in reality. Therefore, after simulations small model vehicles are used to facilitate the testing of new algorithms.

Robust lane keeping is a fundamental property of a research platform for a wide range of traffic scenarios. Any high-level traffic control or maneuvering algorithm assumes that the vehicle can robustly and autonomously remain on its target lane. Therefore, implementing a robust lane keeping algorithm is a prerequisite for testing algorithms in traffic control or vehicle maneuvers.

In this paper, we compare the performance of a well-designed image processing and control pipeline for lane keeping against a straight-forward naive implementation of minimal implementation effort. We show that by using the more complex pipeline, the vehicles will drive more robust and hence can use higher speeds while driving smoothly along the lane.

The employed vehicle platform is a realistic model car in 1:10 scale with a two-axle chassis with Ackermann steering [3] that carries a Raspberry Pi as the compute unit, a servo motor for steering, a DC motor for acceleration and a forward facing Raspberry Pi camera as its main sensor. The test track the cars drive on consists a yellow middle line (ref. Fig. 2), which, together with the car's cameras, is used to guide the cars along the predefined track.

The remained of this paper is structured as follows: The system model used when modelling the vehicle, as well as the experimental environment used when testing the pipeline and control algorithms is described in Sec. II. Afterwards, both processing pipelines are described and simulation results are presented. Sec. IV discusses the measured results and finally, a conclusion is given.

II. System Model

The kinematic equations of motion for a two axle vehicle are commonly represented by the bicycle kinematic model [4]. This model is represented by the following nonlinear state space representation:

$$\dot{x} = v\cos(\theta), \tag{1a}$$

$$\dot{y} = v\sin(\theta), \tag{1b}$$

$$\dot{\theta} = \frac{v}{l}\tan(\varphi). \tag{1c}$$

The state variables of the model are x, y and θ, where x and y are the cartesian coordinates of the midpoint of the rear axle and θ is the heading of the vehicle. The input variables of the model are v and φ, where v is the velocity of the vehicle and φ is the current steering angle.

In reality, the car's steering angle φ cannot change arbitrarily or jump, which mathematically translates to the requirement of continuity. Therefore, φ is modeled to be the output of a first order low-pass with the configurable time constant T and input u, where u represents the target steering angle.

$$\dot{\varphi} = \frac{(u - \varphi)}{T}. \tag{2}$$

The state of the model vehicle is estimated by the image processing pipeline. This image pipeline processes pictures taken from the front-facing camera of the vehicle and estimates the heading θ relative to the lane and lateral distance e_{fa} from the lane. A subsequent steering controller translates this information to the target steering angle which is fed to the vehicles servo. Fig. 1 represents the block diagram of the control loop. The image processing pipeline and steering controller are described in following sections.

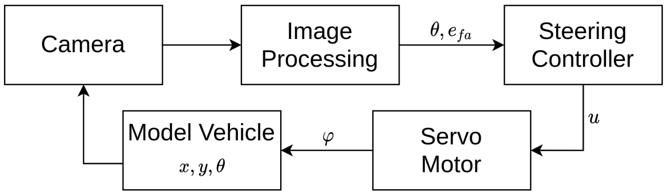


Fig. 1: Graphical representation of control loop

The track and yellow lane line, that the vehicle needs to follow, can be seen in Fig. 2. The track consists of four identical straight and curved sections, forming a closed circuit. When the vehicle is travelling around the track, a positive controller output corresponds to a left turn, while a negative output corresponds to a right turn. Therefore, when travelling clockwise through the track, the target steering angle u is expected to be negative on average.

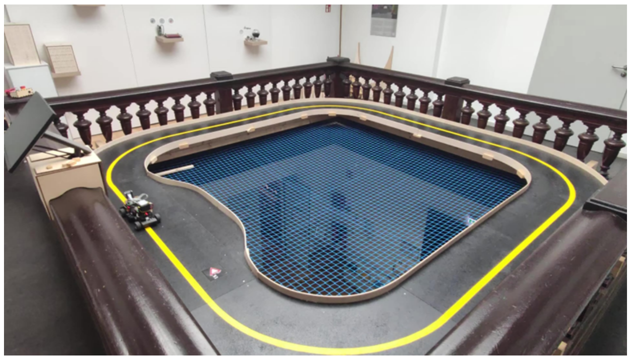


Fig. 2: Track with yellow lane line.

III. Image Processing

A. Naive Implementation

In the most basic implementation of lane detection, all yellow pixels from the incoming camera image are extracted and the biggest grouping is considered to be the lane line. The distance between the x-coordinate of the center of this group and the center of the image is considered the cross-track error e_{fa} and is provided as the input to the steering controller. No heading information θ is extracted from this simple pipeline.

B. Improved Implementation

To improve the naive pipeline, extra steps were added in order to extract additional information from an image, namely the heading of the lane line. The improved pipeline is based on [5]. It is composed of multiple stages in order to calculate the heading θ and cross-track-error e_{fa}. The steps to do so are explained in the following section.

1) Distortion Removal: The start of the image processing pipeline is the distortion removal. For the cars, a fisheye camera was chosen as opposed to a rectilinear camera. The benefit of a fisheye camera arises from its wider angle of view in comparison to a rectilinear camera. However, a fisheye camera inherently has barrel distortion causing straight lines to become curved. The curvature of these lines is dependent on their radial distance from the center of the image. Compensation for this distortion is possible through software, using the process known as camera calibration or camera resectioning [6]. We used the OpenCV library, in particular the `initUndistortRectifyMap` [7] and `remap` [8] functions to accomplish this task. Fig. 3b shows the effect of this stage of the pipeline.

2) Color Thresholding: The next step in the pipeline is color thresholding. This stage removes all pixels from the image that are not the color of the lane line. For this step to be more robust against lighting changes, the image is first converted to the HSV color space from the RGB color space. In order to detect the yellow lane, the pipeline will filter out colors outside of the yellow hue range and then truncate the lower section of the lightness and saturation channels. As a result of this, only bright yellow is left in the image. Fig. 3c displays the effect of this stage of the pipeline.

3) Perspective Transformation: The third part of the image processing pipeline is the perspective transformation [9].

Whenever an image is captured with a camera mounted to the vehicle, the lane line will be trapezoidal as opposed to rectangular, due to the angel the camera is mounted on the car. This perspective requires that all calculations regarding the lane line must compensate for its decreasing width. In order to avoid this requirement, a perspective transformation is employed. For this we use the functions `getPerspectiveTransform` and `warpPerspective` [8] of the OpenCV library.

A consequence of this new perspective, is that the resulting image is a orthogonal view of the track. Fig. 3d displays the effect of this stage of the pipeline.

4) Initial lane detection: After the perspective transformation stage, the start point for the sliding window method in the next step is found. The number of white pixels is counted for each column of the image. The column with the highest number of white pixels is assumed to contain the lane, and is selected as the start point for the sliding window method.

5) Sliding Window Method: In the fifth stage of the pipeline, an analytical approximation of the lane is calculated through the sliding window method [5][10] Using the least squares method [11], a second order polynomial is approximated, creating a best fit through the centers of each sliding window. The polynomial provides an analytical representation of the lane, given by

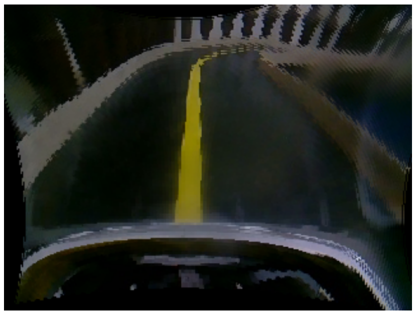
$$g(y) = \beta_0 + \beta_1 y + \beta_2 y^2, \tag{3}$$

where β_i are the coefficients determined using the numpy method `polyfit` [12].

Fig. 3e and 3f display an example of the sliding window method process and the resulting approximated polynomial, respectively.

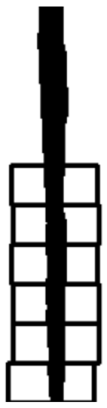
6) Calculation of Heading and Cross-Track-Error: In the final stage of the pipeline, the heading and offset from the lane line are calculated.

(a) Original image captured by the camera on the vehicle


(b) Image after applying the undistortion

(c) Image after filtering out all pixels that don't fit the color of the lane line

(d) Image after warping the perspective to be orthogonal to the track


(e) Visualization of the sliding window method applied to the image

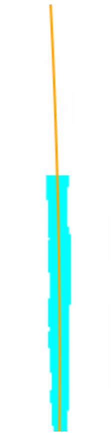

(f) Visualization of the approximated lane line (yellow) with the pixels corresponding to the line (blue)

Fig. 3: Image processing pipeline

An analytical heading is found at a chosen point y along the path with the equation:

$$\theta = \arctan(g'(y)), \tag{4a}$$

with $g'(y)$ being the derivative of the polynomial found in the last stage.

The offset from the lane line is calculated from the fitted polynomial with the following function:

$$e_{fa} = (\frac{w}{2} - g(h)) \cdot s_{mpp}, \tag{5}$$

where w is the width of the image, $g(h)$ is the value of the fitted polynomial at the bottom of the image, s_{mpp} is a scaling constant to translate pixels into meters.

IV. Control Algorithms

A. PID Control

The naive approach uses a PID controller [13] to control the steering of the vehicle. As its input, the controller uses the cross-track-error e_{fa} as calculated by the naive image processing. The controller output is given by

$$u = k_p e_{fa} + k_d \dot{e}_{fa}, \tag{6}$$

where the optimal values for the coefficients k_p and k_d where found by trial-and-error. The I-component of the PID control was intentionally set to zero, as it did not improve the driving robustness. The target steering angle u is sent to the servo to steer the vehicle.

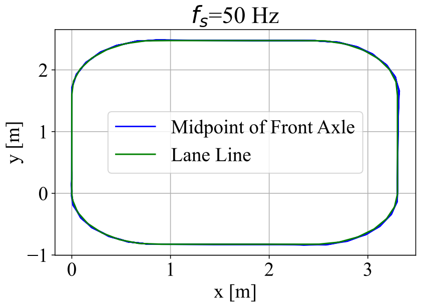
B. Stanley Controller

1) Theoretical Background: The Stanley controller is a non-linear control algorithm that was developed in 2005 by Stanford University with the purpose of controlling a two-axle vehicle using only the heading and the cross track error from the path to be followed. The algorithm has been proven to be asymptotically globally stable for the kinematic model of a two-axle vehicle [14].

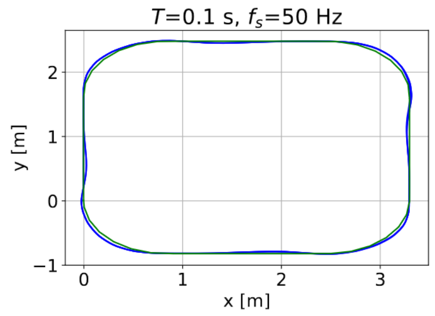
The Stanley controller is a path following controller instead of a trajectory following controller. As a lateral controller, it is designed to keep the vehicle on the path but has no impact on the forward speed. This approach allows for a flexible choice of the vehicle speed, which can be selected as required for a given application, subject to dynamic effects. The Stanley controller is represented mathematically by the following equation:

$$u = \theta + \arctan\left(\frac{ke_{fa}}{v}\right) \tag{7}$$

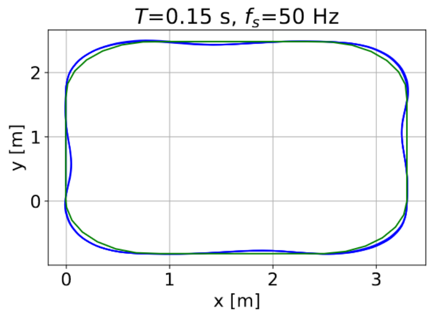
where u represents the controller output, θ is the current heading of the vehicle relative to the path. k is a scaling factor,



(a) Simulated lane keeping of the Stanley controller with a sample rate of 50Hz and instant steering, i.e. $T \to 0$. at a speed of $1.75 m/s$



(b) Simulated lane keeping of the Stanley controller with a sample rate of 50Hz and a time constant of $T = 100 ms$ at a speed of $1.75 m/s$



(c) Simulated lane keeping of the Stanley controller with a sample rate of 50Hz and a time constant of $T = 150 ms$ at a speed of $1.75 m/s$

Fig. 4: Simulated result of Stanley controller applied to vehicle model defined in section II, with varying parameters.

v is the velocity of the vehicle and e_{fa} is the cross track error from the midpoint of the vehicle's front axle to the path.

2) Simulation: To test the suitability of the Stanley controller for our use case we setup a simulation of our scenario. In particular, we were interested up to which steering delay T and sample rate of the system the controller yields acceptable driving performance.

The Stanley controller is modelled in continuous time, which is different from what is used in real-world implementations. The vehicle captures images at a specific sampling frequency, which are then processed by the pipeline before being fed into the controller. Therefore, the behavior of a zero-order hold is simulated. The output of this zero-order hold is then fed into the Stanley controller. This allows for the behavior of the vehicle to be investigated for different sampling frequencies, which allows for a tolerance to be set for the processing speed of the pipeline. The output of the simulation with discrete time processing, sampled at 50 Hz, is shown in Fig. 4. As visible, with a sample rate of 50Hz the controller steers nicely around the track.

Secondly, the behavior of the Stanley controller when subjected to dynamic effects on the steering angle is of great importance, as the global asymptotic stability of the controller was only proven on the kinematic two-axle vehicle model [14]. Therefore, a configurable parameter of the simulation is the time constant T describing the dynamics of the steering servo. The output of the simulation with time constants T 100 ms and 150 ms are shown in Figs. 4b and 4c, respectively. T is shown to have considerable influence on the behaviour of the Stanley controller. As we measured the actual time constant of our steering servo to be in the range of 100-150ms, the simulation results indicate that lane following is possible, however slight deviations from the track and oscillations are to be expected.

V. Results

A. Maximum Forward Velocity

Using the naive pipeline and PID controller on the vehicle, velocities exceeding 1.5 m/s result in reduced robustness. The naive pipeline and PID controller cause the vehicle to oscillate when an outlier in the calculation of the cross-track-error occurs and requires more time to return to the lane. The magnitude of these oscillations increases with the forward velocity of the vehicle. If the velocity exceeds 2.0 m/s, the PID controller crashes the vehicle against the track wall. Even velocities between 1.5 and 1.75 m/s require supervision and manual emergency stops to avoid crashes.

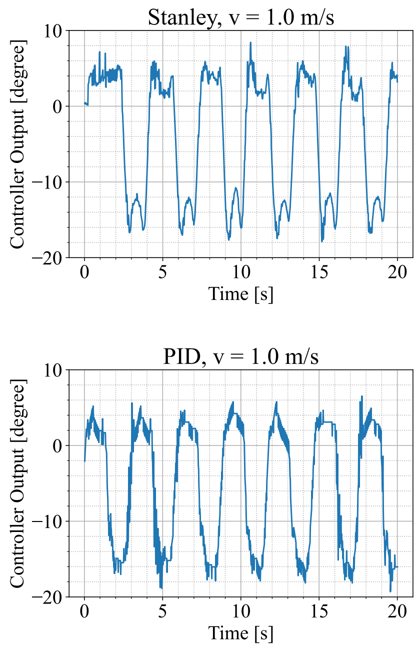
In contrast, the improved pipeline and Stanley controller require no supervision until the forward velocity of the vehicle reaches 2.3 m/s. However, at higher velocities the vehicle oscillates about the path, never converging onto the path. This behaviour has been also foreseen from the simulation results.

B. Controller Output

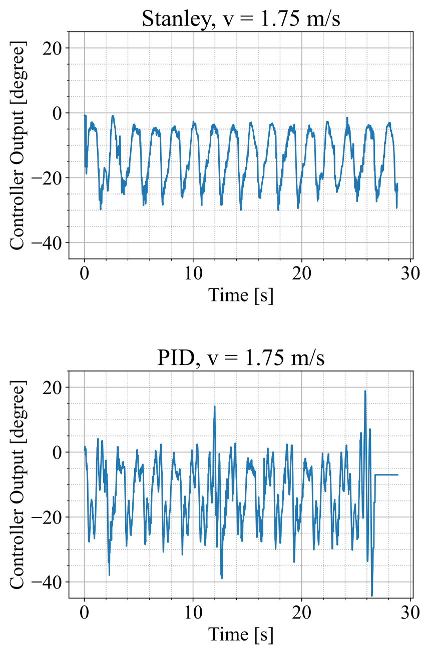
1) Controller Output over Time: Fig. 5a shows the output of the two controllers over time at forward velocity of 1.0 m/s. At low velocity the two controllers have similar behaviour. As the forward velocity of the vehicle increases, the controller output scales appropriately. Between forward velocities 1.0 and 1.5 m/s, the signal output of the two controllers are similar, however at 1.75 and 2.0 m/s the sinusoidal nature of the PID controller degrades, while the sinusoidal nature of the Stanley controller increases.

At 1.75 m/s, the naive pipeline and PID controller can still control the vehicle onto the lane, however as can be seen in Fig. 5b, outliers are more common than with the Stanley controller. These lead to high oscillations in the controller output. When the naive pipeline incorrectly identifies the lane, the PID controller has difficulty returning the vehicle to the lane. This can also lead to the controller crashing the vehicle into the wall, which can be seen at the right end of the graph. In contrast, the improved pipeline and the Stanley controller have no issue following the path at the same velocity.

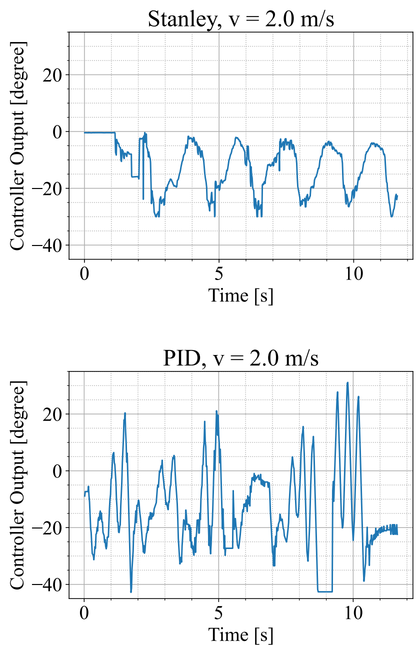
At 2.0 m/s, the PID controller is unable to robustly control the vehicle. As can be seen in Fig. 5c, the output varies between -30 and 30 degrees and there is no periodicity. In comparison, the Stanley controller has a stable oscillation



(a) For v=1.0m/s, both controllers exhibit a periodic and rather smooth steering behaviour.



(b) For v=1.75m/s, the PID controller yields a more hectic steering pattern, whereas the Stanley controller keeps a smooth behaviour.



(c) For v=2.0m/s, the PID controller lets the vehicle oscillate around the lane and eventually crashes against the wall, whereas the Stanley controller still shows a regular and smooth steering pattern.

Fig. 5: Controller output of PID and Stanley controller over time for different forward velocities v.

between -30 and 0 degrees and is periodic, which corresponds to a smooth driving behaviour.

Outliers do not lead to high oscillations when the vehicle is being controlled by the Stanley controller. It has been observed that the vehicle will reorient itself onto the path much quicker than the PID controller.

Due to the relatively short straight section of the track, both controllers are unable to converge to a steady state before the next curve.

VI. Conclusion

In this paper we have compared two approaches on lane keeping for model cars. Our results show that a more elaborate processing pipeline including sophisticated image processing and steering control can significantly enhance the robustness of model cars in lane keeping scenarios compared to a simple PID control implementation. With the presented Stanley controller pipeline, our model cars could reliably steer at speeds of 2.3m/s which is sufficient for many traffic scenarios. Therefore, the presented pipeline serves as a basis for subsequent tests of e.g. overtaking or merging traffic scenarios with the employed model cars.

References

[1] J. Pohlmann, M. Matthe, T. Kronauer, P. Auerbach, and G. Fettweis, "Ros2-based small-scale development platform for ccam research demonstrators," *IEEE Vehicular Technology Conference*, vol. 2022-June, 2022, ISSN: 15502252. DOI: 10.1109/VTC2022-Spring54318.2022.9860981.

[2] S. H. Leilabadi, N. Katzorke, M. Moosmann, and S. Schmidt, "Systematic test case design for autonomous vehicles," *2020 IEEE 23rd International Conference on Intelligent Transportation Systems, ITSC 2020*, 2020. DOI: 10.1109/ITSC45102.2020.9294389.

[3] P. Ge, L. Guo, and J. Chen, "Electronic differential control for distributed electric vehicles based on optimum ackermann steering model," *2021 5th CAA International Conference on Vehicular Control and Intelligence, CVCI 2021*, no. Cvci, pp. 1–6, 2021. DOI: 10.1109/CVCI54083.2021.9661256.

[4] P. Polack, F. Altché, B. d'Andréa-Novel, and A. de La Fortelle, "The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles?" In *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017, pp. 812–818. DOI: 10.1109/IVS.2017.7995816.

[5] A. Sears-Collins, *The ultimate guide to real-time lane detection using opencv*, [Online; Stand 11. Mai 2023], 2021. [Online]. Available: https://automaticaddison.com/the-ultimate-guide-to-real-time-lane-detection-using-opencv/.

[6] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[7] OpenCV, *Fisheye camera model*, [Online; accessed 19-June-2023], 2022. [Online]. Available: https://docs.opencv.org/3.4/db/d58/group__calib3d__fisheye.html.

[8] ——, *Fisheye camera model*, [Online; accessed 19-June-2023], 2022. [Online]. Available: https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html.

[9] R. Szeliski, *Computer vision: Algorithms and applications*. Springer Nature, 2022.

[10] K. C. Bhupathi and H. Ferdowsi, "An augmented sliding window technique to improve detection of curved lanes in autonomous vehicles," in *2020 IEEE International Conference on Electro Information Technology (EIT)*, 2020, pp. 522–527. DOI: 10.1109/EIT48999.2020.9208278.

[11] J. Gergonne, "The application of the method of least squares to the interpolation of sequences," *Historia Mathematica*, vol. 1, no. 4, pp. 439–447, 1974, ISSN: 0315-0860. DOI: https://doi.org/10.1016/0315-0860(74)90034-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0315086074900342.

[12] NumPy, *Numpy polyfit*, [Online; Stand 19. Juni 2023]. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html.

[13] L. Wang, "Basics of pid control," in *PID Control System Design and Automatic Tuning using MATLAB/Simulink*. 2020, pp. 1–30. DOI: 10.1002/9781119469414.ch1.

[14] G. M. Hoffmann, C. J. Tomlin, M. Montemerlo, and S. Thrun, "Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing," in *2007 American control conference*, IEEE, 2007, pp. 2296–2301.