Graphical Simulation of Ackerman Wall Following

Tareq Dandachi, Jason Daniels, Levi Gershon

May 12, 2020

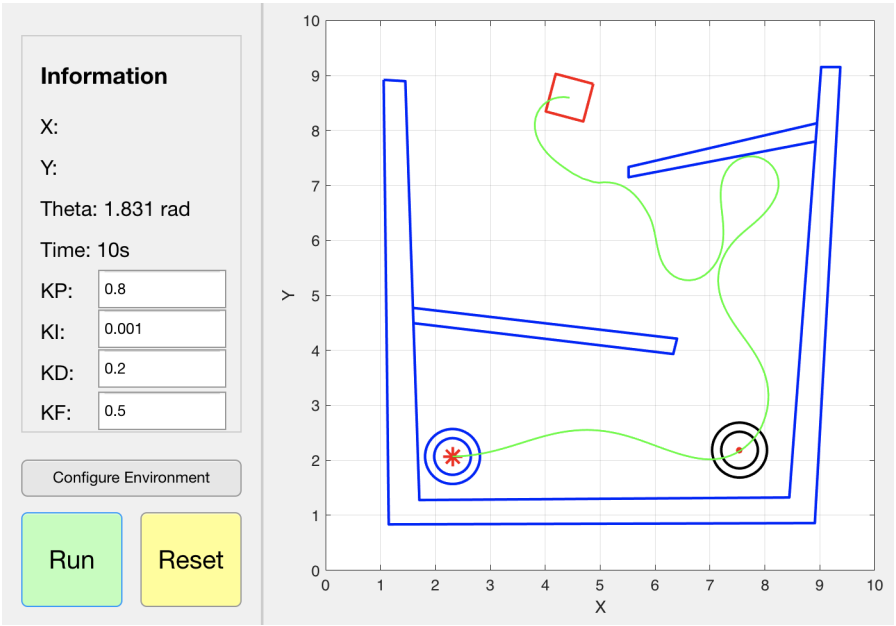


Table of Contents

1 Introduction

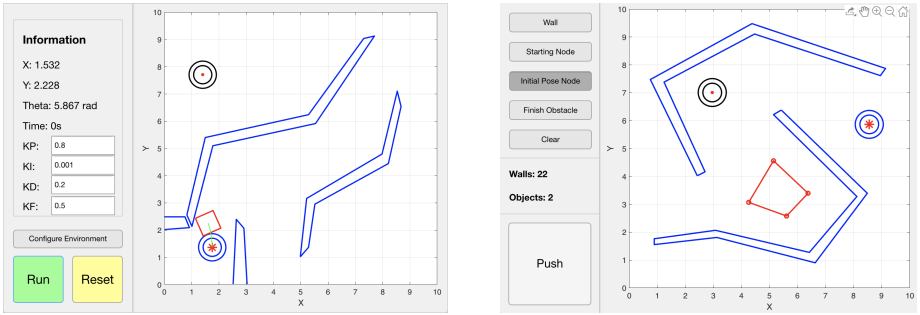
While a trivial task for the human brain, wall following is a far more difficult task for automated systems to accomplish reliably. Nevertheless, it also a relatively well-solved controls problem by this date, and thus constitutes an interesting example of a variable set-point control algorithm. To that end, we have created a graphical simulator of a PID-controlled wall-following robot. It includes an interface to setup a room with a set of obstacles, along with a robot and a specified initial pose. Then, given a set of user-chosen PID coefficients, it will attempt to follow this wall indefinitely.

This graphical presentation allows both for experimentation with the sensitivity of the PID-algorithm used to various parameters, and, through end-user customization, of other control schemes. It can readily be used to observe dependency on the PID coefficients, on initial pose, LIDAR splicing parameters, and on the environmental setup. Furthermore, the classes of the code are heavily abstracted, and can easily be used customized and implemented to experiment with other setups, robot configurations, and control algorithms. To that end, the project is entirely open-source on GitHub, available at the following hyperlink:

https://github.com/tareqdandachi/2086-WallFollower

2 User Interface

The visual nature of the project lends itself to a graphical interface that allows full customization of environmental features and control parameters. As such, the simulator was designed with interactivity and ease of experimentation at the forefront. The primary user interface acts as a sort of central nervous system for the program, managing the system clock and simulation loop while allowing the user to run, pause, reset, or edit the control parameters of the wall-follower at will. Additionally, the primary UI displays information concerning the current simulation, such as the robot's coordinates on the xy-plane, time elapsed, heading angle, and most importantly, a live plot of the robot's path through the environment. An example of this interface running can be seen in figure 1(a).



(a) Primary User Interface

(b) Environmental Configuration Interface

Figure 1: Screenshots of the UI

With use of a secondary environmental configuration interface, seen in figure 1(b), which is accessed through the primary, the program also allows the user to easily customize the layout of the environment in which the wall-follower will operate. This includes the ability to construct an unlimited number of one or two-dimensional objects for the robot to interact with, define the robot's starting position, and specify its initial heading.

3 LIDAR Array Splicing

In a real system, the LIDAR sensors used often have finite fields of vision (FOV), and indeed, this restriction is actually reflected in the control algorithm used. To that end, we first calculate a virtual equivalent to unlimited FOV LIDAR data, and then splice this into three separate segments, representing three (configurable) sensors on the robot, one forward, and one on either side.

To emulate raw LIDAR data, we consider a set of rays emanating from the robot. As this is discrete computation, we take finite set of angles for these rays off of the heading of the robot θ, each separated by an angle $d\theta$. Next, for each ray $\theta_k = \theta + kd\theta$, we iterate over all walls, which are treated as line segments, and calculate the intersection. In figure 2, these are points A and B. We then compare the norms of $\vec{r}_{OA}$ and $\vec{r}_{OB}$, and write the minimal one to the returned LIDAR array, which we consider to be the distance cloud. For all angles θ_k, this then constitutes a reasonable facsimile of real life 2D LIDAR array data.

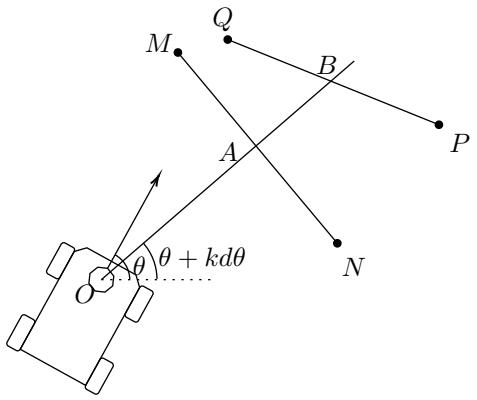


Figure 2: Distance Cloud Algorithm

Next, we take the distance cloud created above, and splice it according to intervals based on the sensor ranges, as defined below:

For instance, $front = distances(-30^\circ \leq \theta_k \leq 30^\circ)$. This creates three (potentially non-disjoint) subsets of the distance cloud, which are together fed into the control algorithm, as elucidated in the following pages.

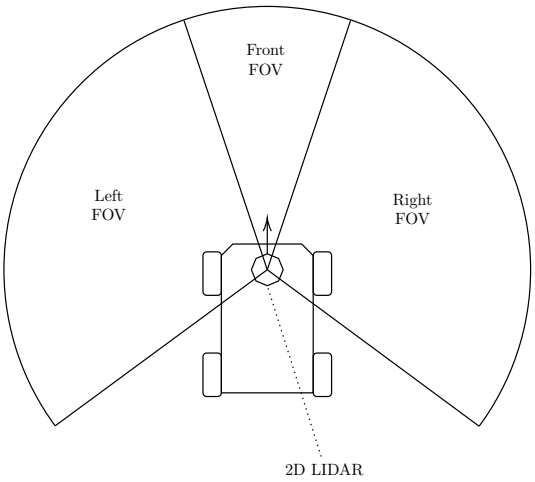


Figure 3: 2D LIDAR Array Splicing

4 Ackerman Geometry and Physics Simulation

The robot we are modeling uses the so called Ackerman steering geometry. The modelled geometric assembly causes the robots drive to be dependent on only two variables, the speed and the angle of turn. This geometry is widely used in robotics due to the way the axles are arranged such that the tyres don't slip when following a path around a curve. The axles are put in an arrangement where all the wheel attachments lie on the radius of a circle and the wheel itself is tangent to the circle at the point of attachment as seen in Figure 4.

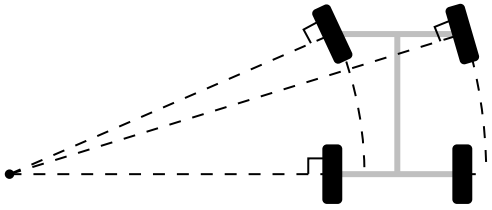


Figure 4: Ackerman Wheel Geometry

The car, having 3 degrees of freedom in a 2D plane, x, y, θ, will be modeled such that it has a constant velocity v to reduce the number of degrees of freedom (given an initial start point) to only one, being θ.

$$\dot{x} = v\cos\theta$$

$$\dot{y} = v\sin\theta$$

To calculate the change in θ, we can use the geometry as shown in figure 5 to find that

$$\dot{\theta} = \frac{v}{L} \tan \alpha$$

where L is the length of the car.

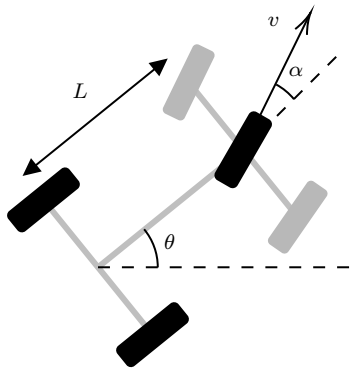


Figure 5: Geometry used to derrive the equation for change in θ

The odometry (state) of the car $\mathbf{x}_t$ at any time t can be modelled as $\mathbf{x}_t = (x_t, y_t, \theta_t, \alpha_t)$ and $\mathbf{x}_t = \mathbf{x}_{t-1} + \Delta\mathbf{x}$ st. $\Delta\mathbf{x} = (\dot{x}, \dot{y}, \dot{\theta}, \Delta\alpha)$. The value of α is dependent on the output of a controller, while all other variables are dependent on the value of α, hence the controller is essential to moving the robot anywhere on the map.

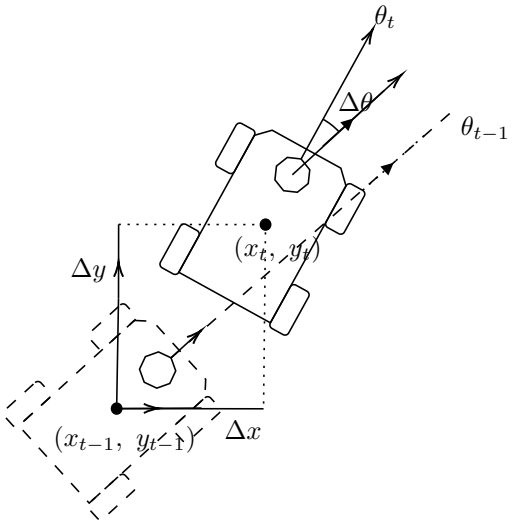


Figure 6: The change in odometry between two timesteps $t-1$ and t

5 PID Controller and Wall Following

Now that we have one variable that controls the whole system, we need a robust method to control its value such that the robot is responsive. In our project, the design modeled is a PID controller that tries to minimize an error e, where e uses the spliced LIDAR scans to estimate how far away the robot is from the wall and try to get the distance to equal 1 unit.

For a certain distance measurement D away from the wall, the error $e = 1 - D$, our steering angle α is defined to equal the amount of error e. The introduction of a PID controller and some heuristics makes the steering angle α more interesting such that

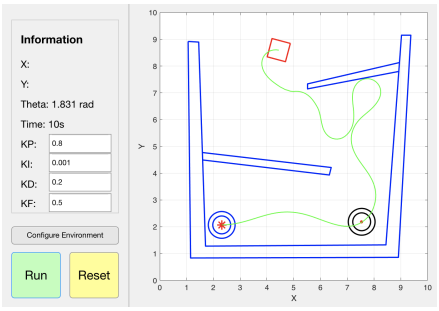
$$\alpha_e = K_p e_t + K_i \int_{t_i}^{t} e_x dx + K_d \frac{de_t}{dt} + [K_f e_f \iff d_f < c_f]$$

where K_p, K_i, K_d and K_f are constants that can be modified by the UI, e_t is the error at time t and e_f is error calculated in a similar fashion to e but using the front LIDAR cloud. The value of $K_f e_f$ is only accounted for when the distance d_f from the front of the robot to the wall is below a certain threshold c_f, which is hard-coded in 'Controller.m'.

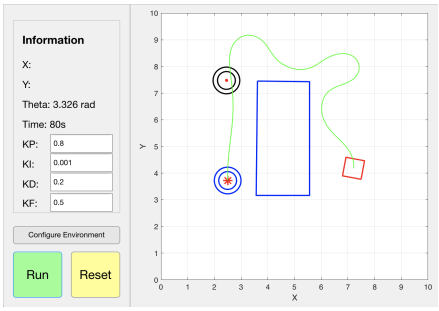
As mentioned previously, the angle α is controlled by the error, which means that α is dependent on one side of the LIDAR at any time instant t, such that the frame of reference for the distance from the wall is either the left or right cloud. We account for that by hard-coding a value in 'Robot.m' that controls whether the robot follows the right or left wall. Hence the actual steering angle α taking into account the side being follow s is $\alpha = s\alpha_e$ where $s = -1$ if the side is right or $s = 1$ if it is left. This definition follows the widely used coordinate conventions in robotics where the x-axis of the car points in the left direction, the coordinate system is left-handed.

6 Results and Conclusion

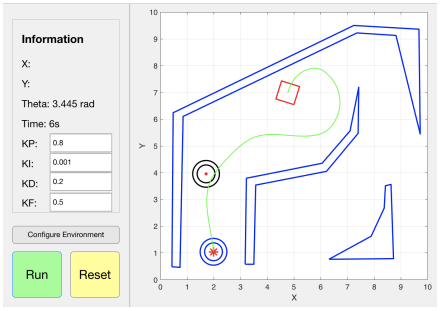
The final product of our project is an interactive environment in which users can easily experiment with the behavior of a PID-based wall-following algorithm in various custom two-dimensional environments. Several examples of the program in use, after having run for some time, are given in figure 7. Aside from the ample learning opportunities this offers, the ability to customize the PID controller parameters allows users to utilize the simulated environment in order to tune a real robot's performance in a model environment. Additionally, the abstracted structure of our program leaves room for users to implement more complex steering geometries, controllers, and even sensor arrays to suit their needs.



(a) Robot Behavior in a Linear Environment



(b) Robot Behavior in an Open Environment



(c) Robot Behavior in a Constricted Environment

Figure 7: Robot behaviour in different environments