Project Phase 4: Control

1. Understanding Open-Loop Control

Before we implement our closed loop controller, we need to understand how open loop control works. An open loop controller is one in which the input is predetermined using your system model, and not adjusted at all during operation. To design an open-loop controller for your car, you would select as your input, u[n], the PWM duty-cycle value that corresponds to your target wheel velocity. You found both of these in System ID. To implement this controller, you would hard-code this input value, as it doesn't change at all as your car runs.

This works great if your model is perfect, and your car is never bumped or otherwise disturbed. In other words, with no mismatch between the idealized behavior we model in equations (1) and (2), you car will drive straight. Model mismatch inevitably arises because every model is ultimately an approximation (recall how you used least-squares to derive your model coefficients θ and β: were your velocity vs. PWM duty-cycle plots perfectly linear?). Since we are assuming ideal characteristics, we will not adjust our input as we go to account for environmental factors or model mismatch (i.e., we have no feedback).

2. Choosing Our Input Value

How do we choose the correct input value for our open loop controller? First, we need to identify a variable (and a value for it) which correspond to our desired system behavior. Once we do that, we need to determine the relationship between the input and this variable.

We will define our control variable, δ, to be the **difference in *distance traveled*** between the left and right wheels.

$$\delta = d_L[k] - d_R[k]$$

Our goal is for the car to **drive straight.** This means the ***velocities***, not necessarily the positions, of the wheels need to be the same,

If we want $v_L[k] - v_R[k]$ to be zero, what condition does this impose on δ?

3. Jolt Calculation

As you (should have!) observed in the System ID lab, the individual motors begin moving at different PWM duty cycles. Recall that PWM is used to digitally change the average voltage delivered to a load by varying the duty cycle (the proportion of a given cycle period for which the power source is turned on). If the cycle period is small enough, the on-off switching is imperceptible, but the average voltage delivered to the load changes proportionally with the duty cycle. Hence, changing the duty cycle corresponds to changing the DC voltage supplied to the motor. If the motors need different voltages to start moving, it is evidence of one or more of the following:

- **Difference in motor parameters.** Your motors might have different armature resistances, which in turn limits the amount of armature current flowing through the motor for a given voltage. If the resistance is higher, the motor will need a higher voltage to force enough current through the internal motor circuit for the rotor to turn.
- **Difference in motor efficiencies.** This could be interpreted as a subsection of the previous item, but the distinction between the two is that we will interpret the first as primarily a consequence of differences in electrical characteristics, and this one as a a consequence of differences in mechanical characteristics. The friction within the motor may be higher for one motor than the other, or perhaps a gear is slightly misaligned in one motor, etc.
- **Mass imbalance in the car.** If the mass of the car isn't distributed evenly between the wheels, the torque required for the wheels to begin turning will differ.
- **Undesired circuit loading.** If one of your motors only starts moving toward the top of the PWM range, you may have somehow added a resistance to the path between your 9V source, your motor, and ground. Check your circuit if this is the case. Alternatively, if your motors are significantly mismatched, consider perhaps adding a (very small — the motor stops moving altogether for an added resistance of 10Ω!) resistance to the faster motor's circuit. You may have to get a bit creative with this if it proves necessary.

Most of the cases outlined above are expected, and we are able to correct for them by modifying the jolts we apply to start each wheel when the car starts up.

Closed-Loop Control of S1XT33N

Introduction

In the System ID lab, you linearized the car system around your operating point (the value of v^* you selected). We will now be deriving the equations which govern the behavior of our system. These equations determine the dynamics of the system that we want to control. We will implement a state-space controller by selecting two dimensionless positive coefficients, $k_l > 0$ and $k_r > 0$, such that the system's eigenvalue is placed in a stable position. (Recall that for discrete systems, eigenvalues must be within the unit circle on the complex plane for the system to be stable).

Deriving the closed-loop model

Previously, we introduced $\delta[k] = d_L[k] - d_R[k]$ as the difference in positions between the two wheels. If both wheels of the car are going at the same velocity, then this difference δ should remain constant, since no wheel will advance by more ticks than the other. We will consider a proportional control scheme, which introduces a feedback term into our input equation in which we apply gains k_L and k_R to $\delta[k]$ to modify our input velocity at each timestep in an effort to prevent $|\delta[k]|$ from growing without bound. To do this, we will modify our inputs $u_L[k]$ and $u_R[k]$ accordingly:

We begin with our open-loop equations from last week:

$$\begin{aligned} v_L[k] &= d_L[k+1] - d_L[k] = \theta_L u_L[k] - \beta_L \\ v_R[k] &= d_R[k+1] - d_R[k] = \theta_R u_R[k] - \beta_R \end{aligned} \tag{1}$$

We want to adjust $v_L[k]$ and $v_R[k]$ at each timestep **proportional** to $\delta[k]$. Let's say we want to drive $\delta[k]$ towards zero. If $\delta[k]$ is positive the left wheel has traveled more distance than the right wheel, so relatively speaking, we can slow down the left wheel and speed up the right wheel to cancel this difference (i.e. drive it zero.). Therefore, instead of v^*, our desired velocities are now $v^* - k_L\delta[k]$ and $v^* + k_R\delta[k]$. As v_L and v_R go to v^*, $\delta[k]$ goes to zero.

$$\begin{aligned} v_L[k] &= d_L[k+1] - d_L[k] = v^* - k_L\delta[k] \\ v_R[k] &= d_R[k+1] - d_R[k] = v^* + k_R\delta[k] \end{aligned} \tag{2}$$

In order to bring $d_L[k+1]$ and $d_R[k+1]$ closer to each other, at time step k we set $v_L[k]$ and $v_R[k]$ to the desired values $v^* - k_L\delta[k]$ and $v^* + k_R\delta[k]$, respectively. Setting the open-loop equation for v equal to our desired equation for v, we solve for the inputs u_L and u_R:

$$\begin{aligned} u_L[k] &= \frac{v^* + \beta_L}{\theta_L} - k_L\frac{\delta[k]}{\theta_L} \\ u_R[k] &= \frac{v^* + \beta_R}{\theta_R} + k_R\frac{\delta[k]}{\theta_R} \end{aligned}$$

These are the inputs required to equalize $d_L[k+1]$ and $d_R[k+1]$. Now, we plug these inputs into our original equations for v_L and v_R to complete the model:

$$\begin{aligned} v_L[k] &= d_L[k+1] - d_L[k] = \theta_L\left(\frac{v^* + \beta_L}{\theta_L} - k_L\frac{\delta[k]}{\theta_L}\right) - \beta_L \\ v_R[k] &= d_R[k+1] - d_R[k] = \theta_R\left(\frac{v^* + \beta_R}{\theta_R} + k_R\frac{\delta[k]}{\theta_R}\right) - \beta_R \end{aligned}$$

Now, we will find the eigenvalue of the system. Let's begin by reviewing some selected terms related to state-space control.

- System variable: Any variable dependent on/responding to the input or initial conditions of a system.
- State variables: The *smallest* set of linearly independent system variables such that the values of the members of this set, together with the known functions of these variables that model the system behavior, *completely determine* the value of *all* system variables for all $t \geq t_0$.
- State space: The n-dimensional space whose axes are the state variables (n is the number of state variables)

To study the dynamics, we will first define our state variables. If the goal is to have the car go straight, then our goal should be to maintain equal velocity for the left and right wheels. However, we are measuring the distance travelled by the left and right wheels, so we will implement a controller that drives the difference in those distances $\delta[k]$ down to zero. Let's pick δ as our state variable.

Does the definition of state variables apply to $\delta[k]$.In other words, it the following statement true: Knowing $\delta[k]$ at each time step plus the initial conditions we can uniquely determine the value of other system variables.

Let's now find $\delta[k+1]$ in terms of $\delta[k]$. We can do so by subtracting the two sides of the previous system of equations from each other.

$$\begin{aligned}
\delta[k+1] &= d_L[k+1] - d_R[k+1] \\
&= v^* - k_L\delta[k] + d_L[k] - (v^* + k_R\delta[k] + d_R[k]) \\
&= v^* - k_L\delta[k] + d_L[k] - v^* - k_R\delta[k] - d_R[k] \\
&= -k_L\delta[k] - k_R\delta[k] + (d_L[k] - d_R[k]) \\
&= -k_L\delta[k] - k_R\delta[k] + \delta[k] \\
&= \delta[k](1 - k_L - k_R)
\end{aligned}$$

Key points:

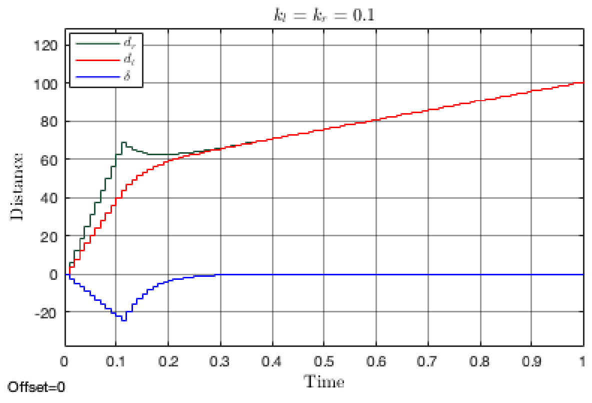
- Since we are trying to drive δ to zero, δ is our state variable.
- Since we have only one state variable, our state space is one-dimensional.
- Since $\delta[k+1] = \delta[k](1-k_L-k_R)$, the coefficient $1-k_L-k_R$ fully describes the evolution of our state variable in time, and we can consider $1-k_L-k_R$ to be our one-dimensional closed-loop descriptor.

4. k-Value Tuning

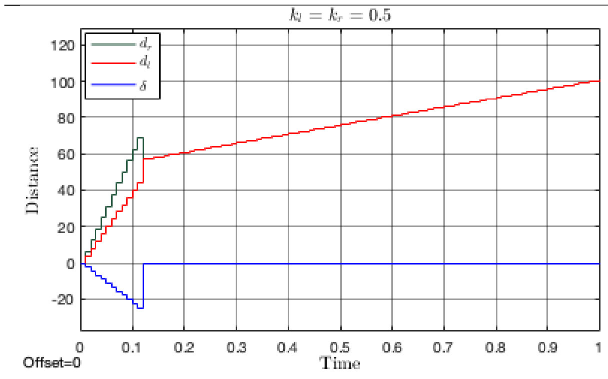
We can get different system dynamic behaviors in time, for different values of $1-k_L-k_R$. Below are some example plots of system behavior (assuming no model mismatch) with various k-values. The rapid ramp-up at the beginning of each plot is from the jolts, which are set to be unequal so that the controller has something to correct.

Note
In this model, it is possible for the distance travelled by a wheel to decrease, but since your car cannot go backwards, this is not possible with your cars. If you are curious about how these plots were made, please see Appendix A.

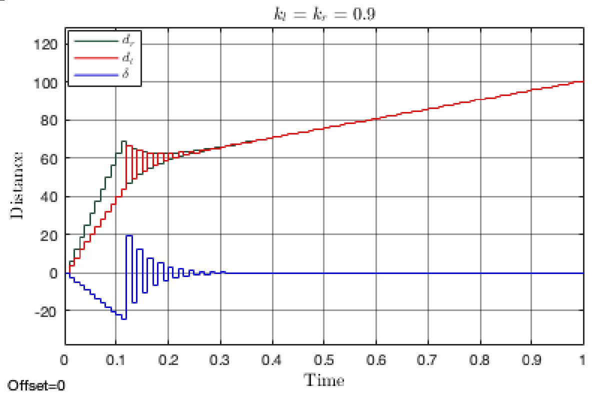
Examples of System Behavior with Various k-Values



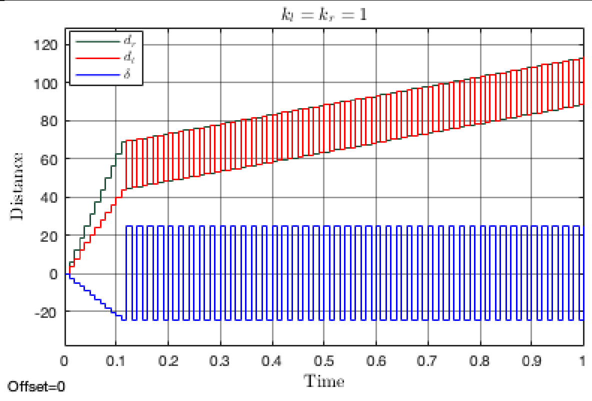
With $k_l = k_r = 0.1$, our system eigenvalue is 0.8, so, since our system is one-dimensional, δ is multiplied by 0.8 at each timestep, slowly driving δ to zero. Since our k-values are equal, the d_l and d_r lines approach each other at equal rates.



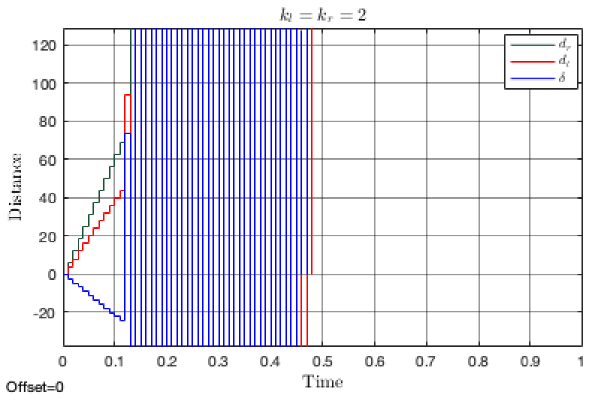
With $k_l = k_r = 0.5$, our system eigenvalue is zero, and since our system is one-dimensional, this means it takes one timestep to send δ to zero.



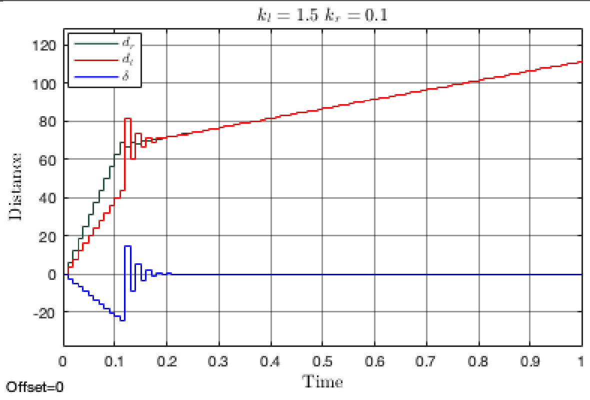
With $k_l = k_r = 0.9$, our system eigenvalue is -0.8, and since our system is one-dimensional, δ is multiplied by -0.8 at each timestep. Since the sign of δ changes at each timestep, you see oscillatory behavior.



With $k_l = k_r = 1$, our system eigenvalue is -1. This is the marginally-stable case: δ's magnitude does not decrease, but its sign changes at every timestep. Picture this behavior in a car: if δ switches sign at every timestep, the car is veering from side to side in a sinusoidal pattern while going forward.



With $k_l = k_r = 2$, our system eigenvalue is -3. This case is oscillatory, like the previous case, but this case is unstable, since δ is multiplied by -3 at each timestep.



With $k_l = 1.5, k_r = 0.1$, our system eigenvalue is -0.6, so while the behavior is oscillatory, δ is still driven to zero and the system is stable. However, one of the wheel controllers (the left one, in this case) is working much harder than the other, which means that the left wheel will need to be able to reach a greater range of velocities than the right wheel

5. Steady-State Error Correction

Ideally, our system should be able to drive δ to 0 over an infinite timespan. However, due to mismatch between the physical system and the model, some steady-state error will be present. We will approximate this value $\delta[\infty]$ by using the final value of δ your car exhibits at the end of a run after your controller has been implemented.

Sanity check question: What would the path of a controlled car with nonzero steady-state error look like? How is this different from the path of a controlled car with zero steady-state error?

Let's assume there is only a mismatch between β in the our model and reality. i.e. β^* is the real model and we mistakenly are using β.

$$v_L[k] = d_L[k+1] - d_L[k] = \theta_L \left(\frac{v^* + \beta_L}{\theta_L} - k_L \frac{\delta[k]}{\theta_L} \right) - \beta_L^*$$
$$v_R[k] = d_R[k+1] - d_R[k] = \theta_R \left(\frac{v^* + \beta_R}{\theta_R} + k_R \frac{\delta[k]}{\theta_R} \right) - \beta_R^*$$

We can re-write $\delta[k+1]$ in terms of $\delta[k]$:

$$\delta[k+1] = (1 - k_L - k_R)\delta[k] - [\beta_L(\frac{\beta_L^*}{\beta_L} - 1) - \beta_R(\frac{\beta_R^*}{\beta_R} - 1)]$$

Just to get a sense of the magnitude of the second term in the previous equation, let's say there is a 10 percent mismatch (i.e. $\Delta\beta_{L(R)}/\beta_{L(R)}$) between model and reality and also $\beta_L = \beta_R + 10$. The second term will become -0.1.

The magnitude of the error is not that large but let's see how it will translate to the steady state error. First let's simplify the model.

$$\delta[k+1] = \lambda\delta[k] + \varepsilon$$

You can show that $\delta_\infty = \frac{\varepsilon}{1-\lambda}$ if $|\lambda| < 1$. You can reach this conclusion if you iteratively expand $\delta[k+1]$, and write $\delta[k]$ in terms of $\delta[0]$. Let's say in our designed system $\lambda = 0.9$. We can see that in this case the small ε can be amplified by a factor of 10 in the steady state. This conclusion was derived based on the assumption that only β has a mismatch. As practice, derive $\delta_[k+1]$ in terms of $\delta_[k]$, when there is a $\frac{\Delta\theta}{\theta}$.

Turning

So far, we have used feedback control to minimize the difference in distance traveled between the two wheels ($\delta[n] = d_L[n] - d_R[n]$). When perturbations cause one wheel to get ahead of the other, the feedback controller adjusts the left and right inputs to correct the error. So, when δ is nonzero, the controller turns the car to send δ back to zero. We can exploit this to make the car turn in a controlled fashion.

Recall that our inputs are:

$$u_L[n] = \frac{v^* + \beta_L}{\theta_L} - \frac{k_L}{\theta_L}\delta[n]$$

$$u_R[n] = \frac{v^* + \beta_R}{\theta_R} + \frac{k_R}{\theta_R}\delta[n]$$

Note that the first term in each input is always constant, while the second changes over time due to feedback.

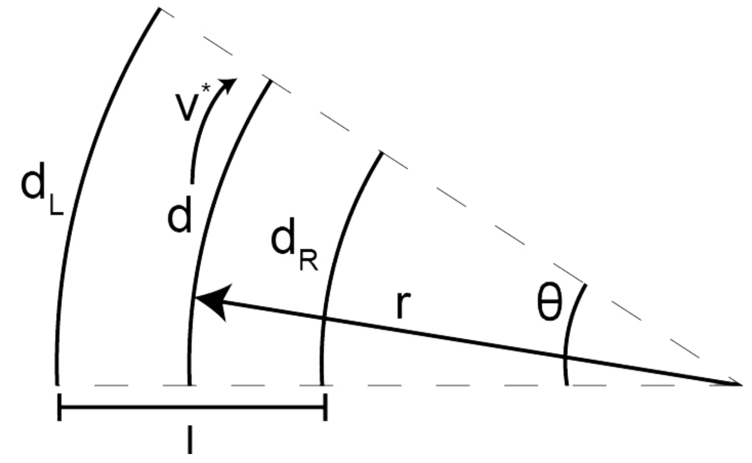
Let's say the right wheel has moved more than the left ($d_R[n] > d_L[n]$), so δ is negative. This causes $u_R[n]$ to decrease and $u_L[n]$ to increase, thereby correcting the disturbance. However, if we artificially increase δ's magnitude by some factor, this would cause $u_R[n]$ to decrease and $u_L[n]$ to increase past the point of straightness: put differently, it causes the car to turn right (if the car's left wheel has travelled further than the right wheel, the car is turning right). Let's explore this idea more formally.

Turning via Reference Tracking

We would like the car to turn with a specific radius r and velocity v^*. To turn smoothly, we want δ to change at a particular rate: rather than just adding a constant offset to δ, we will include a time-dependent reference factor, making δ a function of turn radius r, velocity $v*$, time n, and distance bw the car wheels in centimeters l. Each tick corresponds to approximately 1cm of wheel circumference, so $l = d$.

1. What radius would we use if we want the car to go straight?
2. Inspect the figure below. Write $\delta[n]$ as a function of r, v^*, l, n using the following variables:
   - n - timestep
   - r [cm] - turning radius
   - d [ticks] - distance traveled by the center of the car
   - l [cm] - distance between the centers of the car's wheels
   - ω [radians/tick] - angular velocity
   - θ - angle traveled

Hint: you will find the formula for arc length useful.



Implementing Turns

Before actually implementing turning on the Launchpad, we need to take into account the fact that the sampling periods of our encoders (aka data collection) and our control loop might be different. We'll denote the data collection period by T_d, and the control period by T_c. To see how this fits into your calculated turning reference, note that the units of v^* are ticks/T_d and the units of n are seconds/T_c, or equivalently, ticks: T_c is in seconds, so since we know n is in ticks, we can restate it as seconds/T_c for easier conversion.

T_d is an integer multiple of T_c, and we'll define another variable m as $\frac{T_d}{T_c}$. In our case, $T_c = 100$ms, and $T_d = 500$ms, so $m = 5$. Therefore, **we replace** v^* **with** $\frac{v^*}{m}$, which is in units of ticks/T_c.

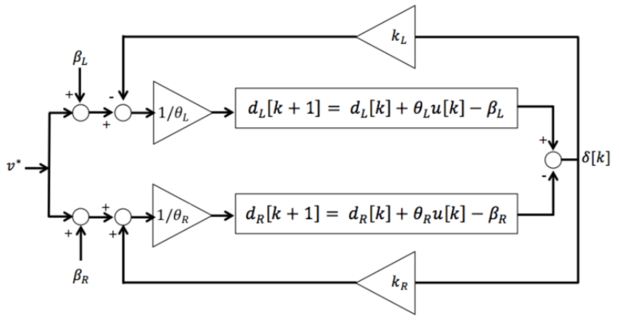
Now you are ready to finish the lab! Go to the Jupyter notebook and follow the instructions to do so.

Appendix A: MATLAB Simulink Model Block Diagram

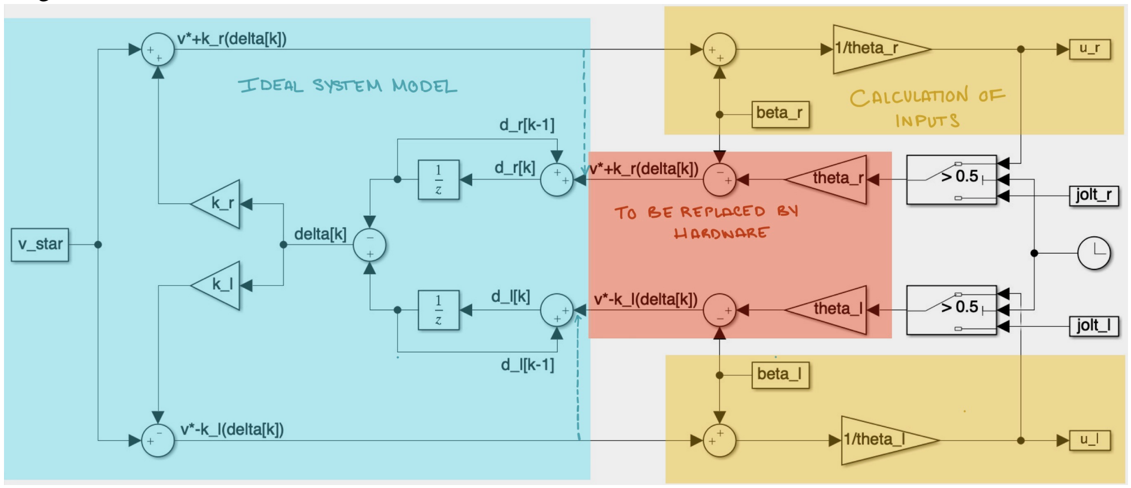
Let's draw a conceptual block diagram of our system. We just need to use blocks and arrows to represent operations and system variables respectively. We can do so by visualizing the system equations.

$$v_L[k] = d_L[k+1] - d_L[k] = \theta_L\left(\frac{v^* + \beta_L}{\theta_L} - k_L\frac{\delta[k]}{\theta_L}\right) - \beta_L$$
$$v_R[k] = d_R[k+1] - d_R[k] = \theta_R\left(\frac{v^* + \beta_R}{\theta_R} + k_R\frac{\delta[k]}{\theta_R}\right) - \beta_R$$

In the figure below, we abstract away the calculation of $u_L[k]$ and $u_R[k]$ and directly plug them into our plants, $d_L[k+1] - d_L[k] = \theta_L u[k] - \beta_L$ and $d_L[k+1] - d_L[k] = \theta_L u[k] - \beta_L$.


k_L
β_L
$1/\theta_L$
$d_L[k+1] = d_L[k] + \theta_L u[k] - \beta_L$
v^*
$\delta[k]$
$1/\theta_R$
$d_R[k+1] = d_R[k] + \theta_R u[k] - \beta_R$
β_R
k_R

We implement the previous diagram in matlab. You can think of the following block diagram as an expansion of the diagram.



$u_L[k]$ and $u_R[k]$ are represented as outputs because they are essentially the outputs of the system you implemented in your code: all the calculations you go through at each timestep lead up to the calculation of $u_L[k]$ and $u_R[k]$, which are then input into the **physical** system via the actuators that generate the PWM signal and make the wheels turn (i.e., the Launchpad's signal pins and the motors/motor controller circuitry). Do not be alarmed by the integrator block: it merely accumulates the velocities to determine distance, and calculates $\delta[k]$ from the differences of the distances. You do not have a direct representation of this integration in your code because the Launchpad stores your distance traveled, but we need it in the block diagram representation because your chosen $v*$, the primary input to the system, is a *velocity*, and is independent of total distance traveled: the block diagram is a direct representation of the system created by setting the equations in (1) equal to their respective equations in (2). Delay blocks in discrete-time systems act like memory overwritten at each timestep, and we can use them to create discrete-time integrators, representing persistent memory, like the one shown in the system above. Keep in mind, $v_L[k]$ and $v_R[k]$ are the *modeled* $v_L[k]$ and $v_R[k]$: according to the equations of our model, this is how $v_L[k]$ and $v_R[k]$ should evolve, and we do not attempt to mathematically model the full complexity of the physical actuator system but instead account for its influence with θ and β.

Notes written by Mia Mirkovic and Meera Lester (2019) Update by Kourosh Hakhamaneshi (2020)