Controlling Physical Systems

0.1 Introduction

Computer programmers are accustomed to instructions that operate *instantaneously* and *reliably*. Both assumptions fail when writing code that controls a physical system. In this tutorial we will take the first steps towards writing programs that can reliably control physical systems in an unpredictable world.

0.2 Open Loop Control

Consider the problem of programming a controller for the self-driving locomotive in Figure 1. In this figure x indicates the starting position of the locomotive and g indicates the goal location.

Our first attempt at developing a controller might look something like the following:

Algorithm 1: Open Loop Control Algorithm

```
def open_loop(x, g):

    # Calculate the distance to travel.
    d = g - x

    # Cover that distance in one second.
    for one second:
        drive forward at a speed of d/second
```

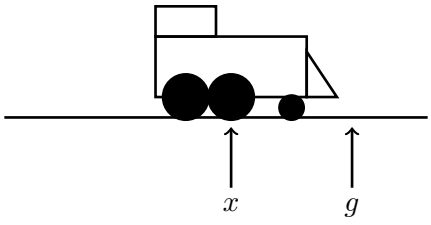


Figure 1: A self-driving locomotive.

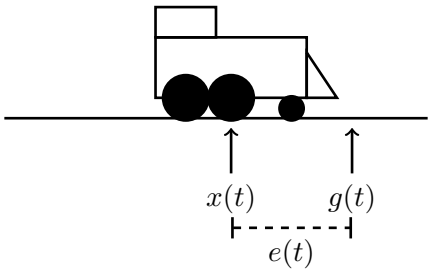


Figure 2: A self-driving locomotive.

Algorithm 1 is an example of an **open loop controller**. Open-loop control involves sending a sequence of control signals that, based on our understanding of the system we are controlling, should move the system into the target configuration.

There are problems with Algorithm 1. First, locomotives are *heavy*. The success of this algorithm relies on the unrealistic assumption that we can instantaneously changing the speed from zero to the desired value of d/s. Second, even after the locomotive reaches the target speed, factors like friction and mechanical imperfections will make it impossible to perfectly maintain that speed. Over time, small errors in speed will result in significant errors in the final position.

0.3 Closed Loop Control

The problems with the naive controller in Algorithm 1 can be avoided by using **closed loop control**. Closed loop controllers continuously monitor the current error in the system and update the control signal to push the error toward zero. This approach tends to be more reliable because the controller responds to the actual state of the system and is able to make adjustments when the system fails to behave as expected.

The PID (**P**roportional, **I**nverse, **D**erivative) controller is the classic example of closed loop control. The next several sections will introduce the PID controller by describing each of the three terms. The following notation, illustrated in Figure 2, will be useful.

$x(t)$ The state of the system at time t. In the case of the locomotive this is the position on the track. More generally, this could describe any state variable that we are interested in controlling. This could be the temperature of a room or the altitude of a rocket.
For now, we will assume that state information is provided by a reliable sensor. In future tutorials we will consider the problem of estimating this value when sensors are absent or unreliable.

$g(t)$ Goal state at time t.

$e(t)$ Error at time t.
We will let $e(t) = g(t) - x(t)$. In the case of the locomotive, this value is zero when the locomotive is at the goal position, positive when the locomotive is to the left of the goal, and negative if the locomotive overshoots and ends up to the right of the goal.

$u(t)$ The control signal at time t. The interpretation of $u(t)$ depends on system we are attempting to control. In some cases $u(t)$ might represent a low-level control signal like the voltage sent to a motor. In other cases we may be working with a robot that allows us to directly specify a desired velocity or acceleration.
In the case of our hypothetical locomotive, we will assume we have an API provides a `throttle` function that takes a number in the range (-100, 100) where +100 represents "full speed ahead" and -100 represents "full reverse".

0.3.1 Proportional Control

Mathematically, we can think of the problem of developing a controller as finding an expression for $u(t)$ in terms of $e(t)$. One simple possibility is to follow the intuition that magnitude of the control signal should be proportional to the current error. In the locomotive example, this means we should apply more throttle when the locomotive is far from the goal location, and ease off as the locomotive gets closer. This idea can be expressed as follows:

$$u(t) = K_p e(t) \tag{1}$$

The value K_p is referred to as a **gain** term. This is a constant that determines how large the control signal will be for a particular error value. Doubling the gain doubles the magnitude of the control signal. Developing a successful controller involves selecting an appropriate gain value, either by analyzing the system or through trial and error.

Algorithm 2 shows how we can implement a proportional controller for the locomotive example.

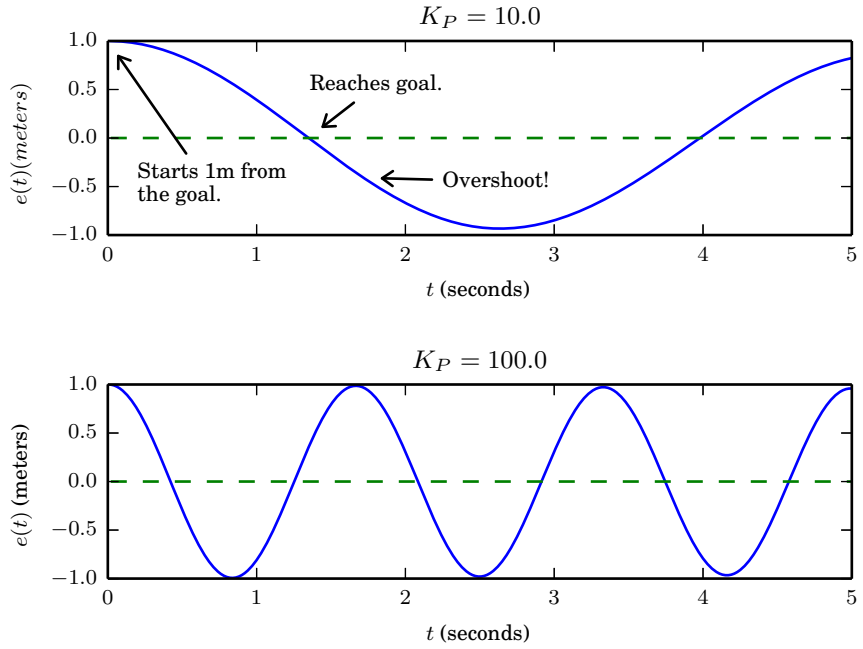


Figure 3: Locomotive position error over time for two different values of K_P.

Algorithm 2: Proportional Control Algorithm

```
def p_controller(train, g, K_P):
    while True:
        e = g - train.x
        u = K_P * e
        train.throttle(u)
```

Figure 3 shows the result of using a proportional controller to implement our locomotive controller using two different values of K_P.

These results are not very satisfying. The locomotive overshoots the goal and then over-corrects with the result that the locomotive drives back and forth indefinitely. Changing the value of K_P doesn't solve the problem. It only changes the period of the oscillations.

0.3.2 Adding a Derivative Term

One problem with our proportional controller is that it only considers the position of the locomotive, not the speed. Intuitively, it seems that if the error is already deceasing quickly we should reduce the control signal to avoid overshooting the goal. Conversely, if the error is still increasing in spite the proportional control, we should further increase the magnitude of the control signal. These intuitions can be captured by adding a derivative term to the controller:

$$u(t) = K_p e(t) + \overbrace{K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t}}^{\text{derivative term}} \tag{2}$$

The term $\frac{\mathrm{d}e(t)}{\mathrm{d}t}$ describes the rate of change in the error. This is negative if the error is decreasing and positive if the error is increasing. The value K_d is a gain that is used to tune the impact of the derivative term.

Using the derivative term requires us to know $\frac{\mathrm{d}e(t)}{\mathrm{d}t}$, but sensors don't usually provide direct access to this value. Instead, we need to estimate it by tracking the change in error over time. Even though the physical systems we are controlling operate continuously, our algorithms necessarily perform their steps at discrete time intervals. Assuming our controller is updated every Δt seconds, we can estimate the derivative as the slope between the two most recent error values:

$$\frac{\mathrm{d}e(t)}{\mathrm{d}t} \approx \frac{e(t) - e(t - \Delta t)}{\Delta t} \tag{3}$$

Discrete approximations like this are common in robotics and in other areas of scientific computing but they aren't always made explicit. It takes some experience to get comfortable moving from continuous to discrete formulations. Algorithm 3 illustrates how we can use the discrete approximation in Equation 3 to implement the control algorithm described in Equation 2.

Algorithm 3: Proportional Derivative Control Algorithm

```
def pd_controller(train, g, K_P, K_D):
    e_prev = g - train.x
    while True:
        e = g - train.x
        dedt = (e - e_prev) / train.dt  # Equation 1.3
        u = K_P * e + K_D * dedt
        train.throttle(u)
        e_prev = e
```

Figure 4 shows the result of introducing the derivative term in our controller. For appropriate values of K_D The oscillations are damped, and the locomotive settles at the goal location.

Stop and Think

- The `while` loop in Listing 3 does not include any explicit delays. It is written under the assumption that methods called on the `train` object will only return after an appropriate delay. What could go wrong if this is not the case? In other words, what will happen if the execution time of the loop is much shorter than `train.dt`?

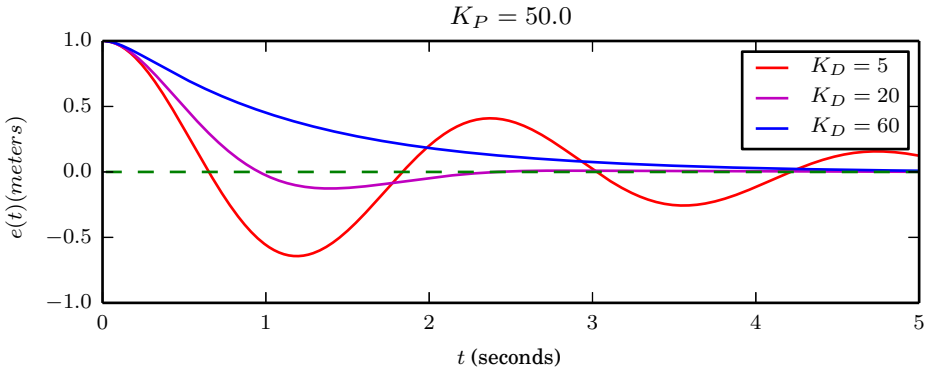


Figure 4: Locomotive position error over time for three different values of K_D.

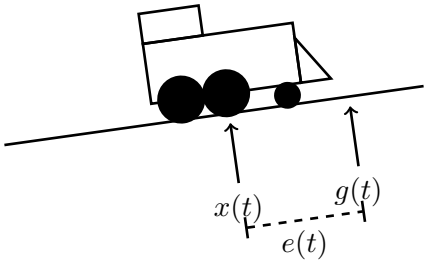


Figure 5: Locomotive on a hill.

Droop

What happens when we try to apply the same controller when the locomotive is located on a slight incline as shown in Figure 5? The results are displayed in Figure 6. In this situation the locomotive never quite reaches the goal position. The locomotive comes to rest at the point where the proportional force applied by the controller is exactly counterbalanced by gravity. Increasing K_p will move the stationary point closer to the goal, but this controller will never drive the error all the way to zero.

The situation where the proportional term is not sufficient to drive the error term to zero is sometimes referred to as "droop".

0.3.3 Adding an Integral Term

The problem of droop can be addressed by adding one more term to our controller:

$$u(t) = K_P e(t) + \overbrace{\boxed{K_I \int_0^t e(\tau)d\tau}}^{\text{integral term}} + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{4}$$

Where the derivative term allows the controller to look forward in time, the integral term allows the controller to look backwards in time. The integral $\int_0^t e(\tau)d\tau$ essentially "stores up"

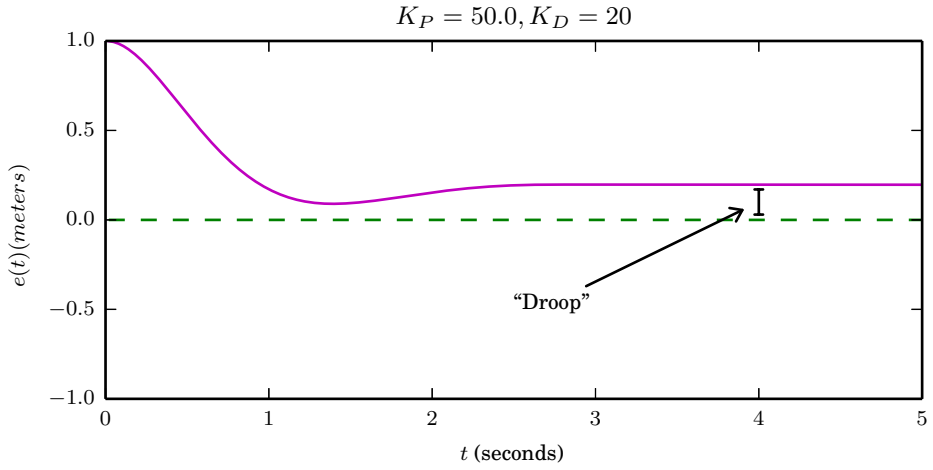


Figure 6: Locomotive position error over time for the PD locomotive controller when the locomotive is on an incline. The locomotive stops short of the goal at the point where the control output is counter-balanced by gravity.

the error that the system sees over time. As long as the error fails to reach zero, $\int_0^t e(\tau)d\tau$ will steadily increase in magnitude.

As with the derivative, the integral value is not available directly, but must be estimated from discrete samples. In this case, the integral can be estimated using a summation that adds the current error value at each time step:

$$\int_0^t e(\tau)d\tau \approx \sum_{i=0}^{t/\Delta t} e(i\Delta t)\,\Delta t$$

$t/\Delta t$: # of steps before time t

$e(i\Delta t)$: Error at time step i

(5)

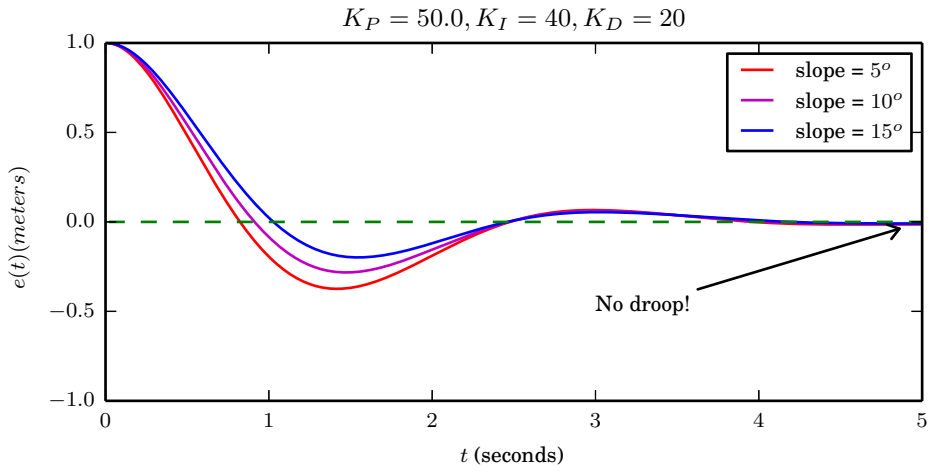


Figure 7: Locomotive position error over time for a different slopes. With appropriate gain values the PID controller reliably moves the locomotive to the goal location.

Algorithm 4: PID Control Algorithm

```
def pid_controller(train, g, K_P, K_I, K_D):

    e_prev = g - train.x
    e_sum = 0              # accumulator for integral term
    while True:
        e = g - train.x
        e_sum = e_sum + e * train.dt    # From Equation 1.5
        dedt = (e - e_prev) / train.dt # Equation 1.3
        u = K_P * e   +   K_I * e_sum   +   K_D * dedt
        train.throttle(u)
        e_prev = e
```

Algorithm 4 illustrates a complete PID controller. Figure 7 shows the result of adding an integral term to our locomotive controller. With an appropriate value of K_I, the resulting controller reliably moves the locomotive the goal regardless of the slope.

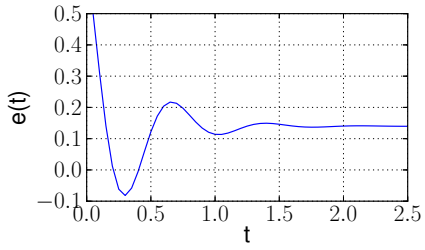
Stop and Think

- Practical implementations of PID controllers often place a cap on the amount of error that can be accumulated in the integral term of the controller. Why do you think this is necessary? What could go wrong if the controller were to start out far from the goal configuration, accumulating a large amount of error before the goal is reached?

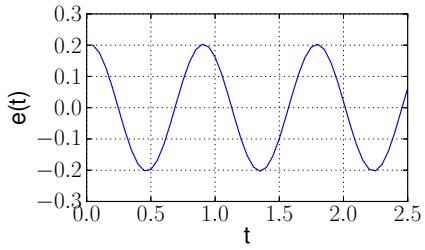
- The process outlined above for tuning the PID gain terms is ad-hoc: we just experimented with different values until the behavior looked right. A more rigorous approach would require a quantitative way to evaluate the success of a particular PID controller.

Can you think of some quantitative measurements that might be useful for comparing two controllers?

- Consider the following graph of error as a function of time. Assuming that this system is being controlled by a PID controller, how would you suggest what the gain terms be modified?



- Consider the following graph of error as a function of time. Assuming that this system is being controlled by a PID controller, how would you suggest what the gain terms be modified?



0.4 Proportional Robot Control

You may feel that a self-driving locomotive is a disappointingly simple robot: all it can do is move backward and forward along a linear track. Don't fear, these tutorials will mostly focus on mobile robots that are able to move freely in multiple dimensions. In this section we will develop a proportional controller for driving a wheeled robot to a goal location as illustrated in Figure 8.

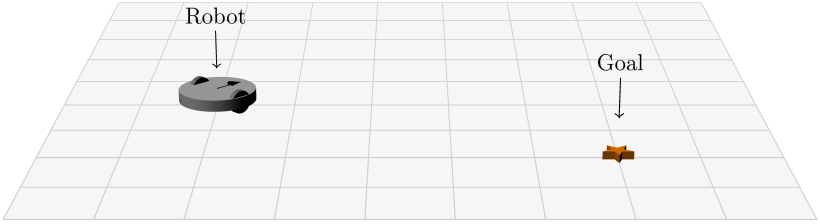


Figure 8: Differential drive robot.

This is an example of a **differential drive** robot. This type of robot has two independently controllable wheels. When both wheels are rotated in the same direction at the same speed the robot moves directly forward or backward. If both wheels are turned in opposite directions the robot will rotate in place. The robot can follow a curved path by rotating each wheel at a different speed.

Since it is not intuitive to steer a differential drive robot by directly controlling the wheel velocities, the driver software for such robots often accepts commands specifying two velocity values: v and w, where v is the forward velocity in meters/second and w is the rotational velocity in radians/second.

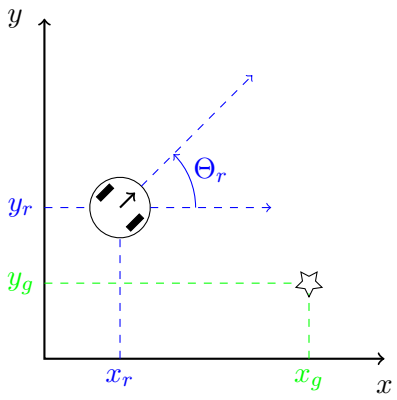


Figure 9: Overhead view of the robot and goal.

Figure 9 illustrates the parameterization of the control problem. The pose of the robot is specified with three numbers: (x_r, y_r, Θ_r), where the x_r and y_r coordinates specify the position of the center of the robot with respect to some fixed point in the room and Θ_r value indicates the robot's heading. Heading values (in radians) are in the interval $[-\pi, \pi]$, where $\Theta_r = 0$ indicates that the robot is pointing along the x-axis with Θ_r increasing as the robot turns counterclockwise. The value of Θ_r in figure Figure 9 is approximately $\frac{\pi}{4}$ (or 45^o).

We can solve this control problem by creating two separate proportional controllers that will operate simultaneously: one controller to output a rotational velocity that turns the robot toward the goal, the other to output a forward velocity that keeps the robot moving forward until it reaches the goal.

The first step in developing the rotational controller is determining the desired angle for the robot. The problem is illustrated in Figure 10. Determining Θ_g from the positions of the robot and goal requires some trigonometry:

$$\Theta_g = \tan^{-1} \frac{y_g - y_r}{x_g - x_r} \tag{6}$$

Actually, this won't *quite* give us what we want. Equation 6 doesn't distinguish between the case when both the numerator and denominator are positive and the case when they are both negative. This means that a goal above and to the right of the robot will result in the same angle as an object below and to the left. The math libraries for many programming languages address this difficulty by providing a function named `atan2` that takes the numerator and denominator as separate arguments and correctly calculates the angle taking the

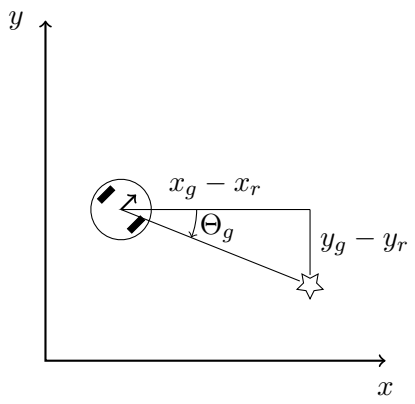


Figure 10: Goal angle calculations

signs of the arguments into account. A Python implementation might look something like the following:

```
theta_g = math.atan2(y_g - y_r, x_g - x_r)
```

Given that we can calculate a target angle, our rotational controller can be specified using the formula for a proportional controller (dropping the time index for clarity):

$$w = K_{P_w}(\Theta_g - \Theta_r) \tag{7}$$

Unfortunately, subtracting angles in a sensible way requires some caution. Consider the situation in Figure 11. Naively calculating the error by subtracting the two angles gives us $.9\pi - (-.9\pi) = 1.8\pi$. Plugging this into our proportional controller will result in a hard left turn. This isn't exactly wrong: it *is* possible for the robot to point toward the goal by turning to the left. However, it clearly makes more sense to select the smaller angle between our current heading and the target heading. We will use the symbol $\ominus$ to indicate an angle subtraction operation that has this effect. With this modification we can completely specify the control scheme for the differential drive robot.

$$w = K_{P_w}(\Theta_g \ominus \Theta_r) \tag{8}$$

$$v = K_{P_v}\sqrt{(y_g - y_r)^2 + (x_g - x_r)^2} \tag{9}$$

Equation 9 expresses the idea that the robot's forward velocity should be proportional to its Euclidean distance from the goal. Since this velocity control is independent of the angle, the robot may initially move away from the goal. We could address through a more complicated control mechanism, but the point here is to illustrate a simple example of proportional control. Figure 12 shows several example trajectories of a simulated differential drive robot under the control scheme presented in Equations 8 and 9.

Notice that, unlike the locomotive example, it wasn't necessary to include integral or derivative terms in this controller. They weren't needed because we were able to able to

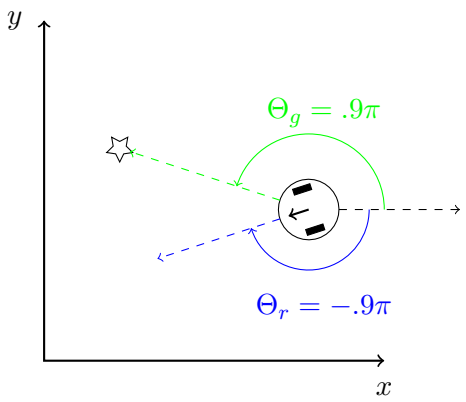


Figure 11: The robot can either turn .2π radians clockwise, or 1.8π radians counterclockwise to face the goal. We want a difference operation that always returns the smaller angle.

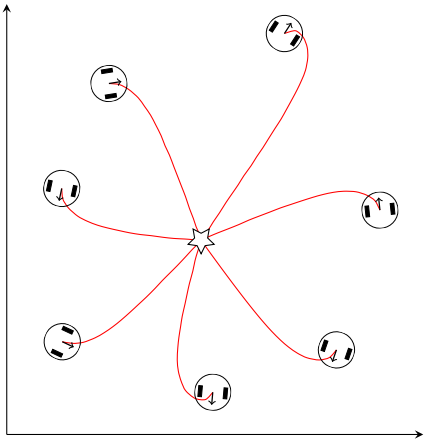


Figure 12: Example trajectories for a proportional controller that moves a differential drive robot to a goal location. The robot images indicate the starting positions for seven simulated navigation trials.

set the speed of the robot directly. This means we didn't need to worry as much about overshooting the goal or applying too little force to reach it. It often makes sense to start with a pure proportional controller and add integral or derivative terms if they prove necessary.

0.5 Limitations of PID Controllers

PID control is widely used because it is simple to implement and can be applied in a wide range of situations. We can use PID control even when we don't have an accurate model of the system we are controlling. In this tutorial we used trial and error to develop a PID controller for a simulated locomotive. It wasn't necessary to understand the physics involved: we didn't need to estimate frictional forces, we didn't need an exact specification of the forces created by the throttle. We only needed to tune the three gain parameters until we were

satisfied with the performance.

The ad-hoc nature of PID control can also be seen as a disadvantage. PID controllers tend to work well in practice, but they don't provide any optimality guarantees. In cases where we *do* have an accurate system model, it's possible to develop more principled controllers that explicitly minimize a cost function related to the quality of the solution. This could allow us to create a controller that reaches the goal in the minimum time or with minimal energy expenditure. Such controllers are beyond the scope of this tutorial, but the classic optimal controller is the **Linear Quadratic Regulator** or LQR.

More significantly, the kind of low-level control discussed in this tutorial is obviously not appropriate for complex control problems that extend over longer periods of time. A PID controller will not save the day for the robot in Figure 13. Problems like this will require planning algorithms of the sort described in later tutorials.

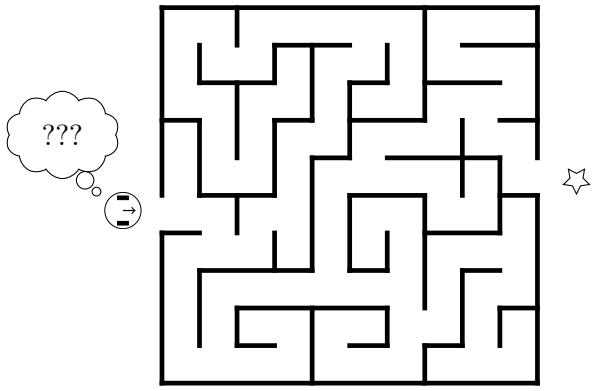


Figure 13: A control problem that can't be solved with the tools in this tutorial.