Chapter 6

Control

Robot behaviors are often expressed with high-level abstract terms, such as “move to a point” or “follow a path”. These behaviors must be accomplished using low-level voltage signals, but the mapping between the high-level behaviors and the low-level signals is often nonobvious, which brings about the need for a *controller*. Given a high-level signal indicating what the robot should accomplish (the input), the controller computes voltage signals for the motors (the output) to try to achieve the desired behavior. In this chapter, we will study different types of controllers and learn how we can use them to realize behaviors that we want a robot to have.

6.1 Principles of Control Design

When designing a controller for a robotic system, the primary considerations are what the inputs to the controller should be, and how those inputs should map onto motor voltages. At the same time, we should consider how complicated we want the system to be. Does our application allow for more simplicity, or should we add more inputs in order to control the system in more intricate ways?

Let us begin by considering a simple case of binary input. Suppose that we have a stationary robot whose only purpose is to monitor when the family cat enters or leaves the house via the cat door. Two signs are located on either side of the robot: one that says “In”, and another that says “Out”. The robot has a one-DoF revolute arm that it uses to point at the signs. Each of the signs is located at one of the arm’s two joint limits, such that when the arm hits a joint limit, it is pointing at a sign.

We could write a very basic controller for this robot that behaves as follows:

- When the cat is observed entering the house, the controller drives the arm at maximum velocity toward the “In” sign, until the joint limit is reached.

- When the cat is observed leaving the house, the controller drives the arm at maximum velocity toward the "Out" sign, until the joint limit is reached.

We call this type of controller a *bang-bang controller* because it switches abruptly between binary states. Another name for bang-bang controller is *on-off controller* because frequently, the binary states being switched between are "on" and "off'. (For example, consider a heating system that turns on when the temperature drops below 72 degrees and turns off when the temperature reaches 72 degrees.)

Bang-bang controllers have the advantage of being easy to implement, but it is clear that they have several disadvantages. In the context of this example, consider that suddenly applying maximum voltage to the motor and driving it until it hits a physical stop will cause wear and tear on the robot. Moreover, suppose that the cat decides to stand in its cat door, halfway in and halfway out of the house, and the robot's perception of the cat's action repeatedly oscillates between entering and leaving. If this happens, the robot arm may swing back and forth rapidly between the two signs, causing further wear and tear and overall undesirable behavior.

To design controllers that are more useful for real-world robotics applications, we will need to leverage inputs that have more than two possible states. We will consider some of these input types and examples controllers in the following list. Note that each kind of input is considered in isolation in the examples, but quite often we will design controllers that use a mix of input types, such as a hybrid force/position controller. Also note that the list is organized such that for each input type, the further down it appears on the list, the farther away it is from the actual output, and thus the harder the control problem. One of the main factors influencing the difficulty of the problem is kinematic constraints on motion. Both holonomic and nonholonomic constraints can prevent motion in forbidden directions, thus impacting velocity control. Even though nonholonomic constraints do not integrate into a position constraint, these constraints can still complicate the job of a position controller because it must overcome the robot's nonholonomic constraints.

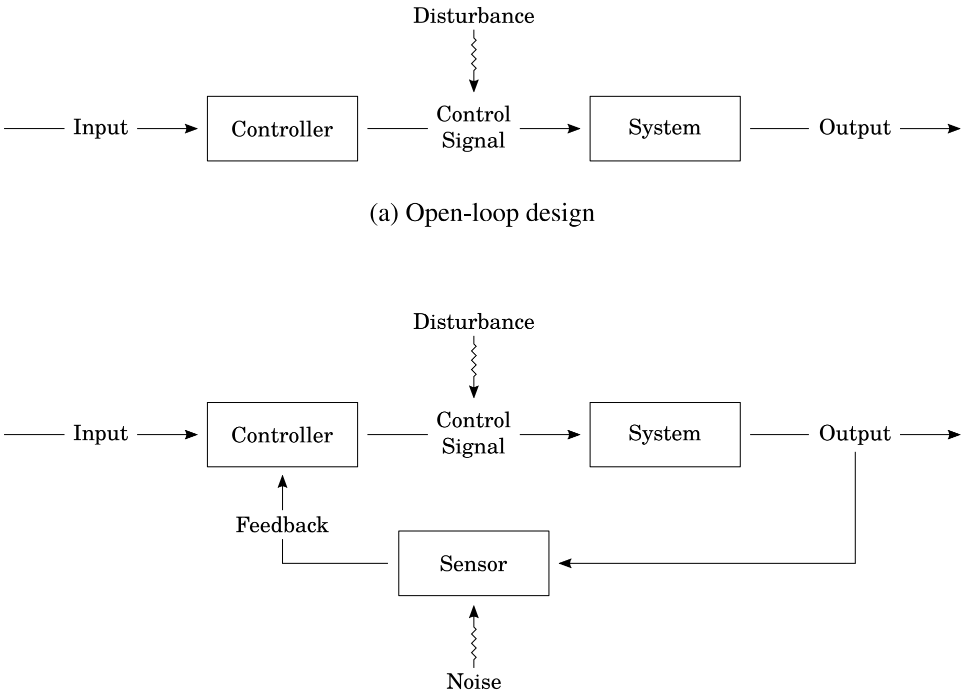
- *Force*. Suppose we have an industrial robot arm that finishes metal objects by lowering a polishing wheel onto them as they go by on a conveyor belt. For this application, we may choose to use *force control* by specifying the desired amount of force we would like the robot to apply while polishing. The controller drives the polishing wheel downward with this much force for a set amount of time and then raises the wheel back to the initial position.
- *Velocity*. Consider an industrial robot arm that traverses back and forth on a straight segment of linear tracks. The robot picks up a payload from one end of the tracks, moves to the other end, drops the payload off, and then moves back to its initial position. The weight of the payload is assumed to always be the same. Suppose that we would like to specify a speed that the

robot should travel at for the duration of the round trip, in which case we may wish to use *velocity control*. The controller uses the velocity input as well as known constants (such as the length of the track, the weight of the robot, and the weight of the payload) to compute the motor speeds necessary to achieve the desired velocity, both for the trip with the payload and the one without, and sends the corresponding voltage signals to the motors.

- *Position*. Suppose we have a quadcopter that needs to autonomously visit a series of waypoints as it inspects a bridge for structural damage following an earthquake. We would likely use *position control* for this problem. The controller takes in a sequence of waypoints (which are positions) as input, including an initial waypoint that is the current position of the robot and a final waypoint that is the goal position. Then the controller determines the voltage signals needed to move the quadcopter from one waypoint to the next waypoint in the sequence and drives the motors with those signals.
- *Pose*. Suppose we are tasked with writing a controller to make an autonomous car parallel park. To complete this task, we would likely choose to use *pose control* (that is, control of both position and orientation) since the initial and final orientation of the car matter. The controller takes in both the current pose of the car and the desired end pose, computes the sequence of motor voltages required to move the car to the end pose (taking into account the constraints of the wheels), and then drives the motors accordingly.

Before we can build any of these controllers, we need to make another important design decision: whether the controller will be *open-loop* or *closed-loop*. Fig. 6.1 gives a diagram for both designs. The difference is that in open-loop design, the output of the system does not factor into the decisions that the controller makes, whereas it does in closed-loop design. Another name for closed-loop control is *feedback control* because the system output that the controller incorporates into its decision-making is called *feedback* and is measured by some appropriate sensor.

Let us return to the velocity control example to illustrate the difference between the two controller designs. If the controller is open-loop, then the actual velocity of the robot during the round trip is not considered by the controller; it is assumed to be correct based on the initial calculations. Of course, it may not be correct if there is a *disturbance* that impacts how the system performs. For example, if the payload is much lighter than the controller believes it to be, then the robot will end up moving too fast. An open-loop controller cannot correct this. In contrast, if the controller is closed-loop, the actual velocity of the robot will be measured and passed as input to the controller, which allows for compensation of disturbances. In the case of a payload that is too light, the controller will detect that the robot is moving too fast and can slow it down until the desired velocity is reached.



(a) Open-loop design

(b) Closed-loop design

Figure 6.1: Block diagrams illustrating the architecture of (a) an open-loop system and (b) a closed-loop system. Note that all of the signals (depicted by arrows) can be affected by noise or disturbances. For example, the control signal could be affected by a disturbance that adversely affects the system output, or readings from the sensor in (b) could be noisy.

Closed-loop controllers do have their share of drawbacks, however. Since they incorporate feedback into their decision-making, they must be designed with more sophisticated control logic, which in general makes them more difficult to build. Moreover, the requirement of additional sensing capabilities adds to the monetary and computational cost of the system. Another issue to consider is that if a closed-loop controller cannot handle all disturbances and noise in an appropriate manner, it may actually exacerbate a problem rather than mitigate it. To illustrate this point further, consider the previous example of a position controller for a quadcopter, and suppose that we use a closed-loop controller that takes in the actual position of the quadcopter as feedback. Now suppose that after executing a plan to fly from waypoint A to waypoint B, the closed-loop controller determines that the actual position of the quadcopter is too far away from waypoint B to be satisfactory, and in an attempt to fix this, the controller overcorrects, resulting in the quadcopter still being too far away. Without careful programming, the controller may make the quadcopter repeatedly overshoot the waypoint and not make any forward progress. An open-loop controller cannot have this problem because it does not use feedback.

Given all of these trade-offs between open-loop and closed-loop design, it is important that we carefully consider what kind of disturbances might affect the system, and how we should (or should not) deal with them. If the disturbances are largely predictable and can be countered with good calibration and a structured environment, then we would likely prefer open-loop design. In contrast, if the tasks the robot is doing and the environment it is operating in can lead to a significant amount of error or high uncertainty, we would prefer closed-loop design in order to reduce the effect of error and achieve the behaviors we would like the robot to have. Take a few minutes now to reread the four controller examples given above and ask yourself whether you would choose open-loop or closed-loop design for each controller, and why. Be sure to consider both the advantages and disadvantages of your choices.

6.2 PID Control

Now that we have distinguished between open-loop design and closed-loop design, we will describe one of the most commonly used closed-loop control mechanisms: the *PID controller*. PID stands for *proportional-integral-derivative*. The basic idea is that a PID controller uses the present error, the past error, and a prediction of the future error in order to determine an appropriate control signal to reduce the error. Mathematically, the general form of a PID controller can be defined as

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}, \tag{6.1}$$

where

$$e(t) = r(t) - y(t) \tag{6.2}$$

is the error in the output, defined as the difference between the desired output r and the measured output y. We see that the control signal $u(t)$ is the sum of three terms:

- the *P-term*, which is proportional to the error,
- the *I-term*, which is proportional to the integral of the error, and
- the *D-term*, which is proportional to the derivative of the error.

Each of these terms has a nonnegative coefficient: K_p, K_i, and K_d, respectively. The latter two coefficients are often written as

$$K_i = \frac{K_p}{T_i} \qquad \text{and} \qquad K_d = K_p T_d, \tag{6.3}$$

because T_i, the integration time, and T_d, the derivative time, both have physical meaning. Then we can set $K = K_p$ and rewrite (6.1) as

$$u(t) = K\left(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}\right). \tag{6.4}$$

We call K the *proportional gain*. As we will later see, it is necessary to *tune* a PID controller in order to determine what constant we should use for the the gain. If K is too high, then the controller may become unstable. Conversely, if K is too low, then the controller may not respond strongly enough to error.

It is important to note that not all PID controllers use all three terms. If one or more of the terms are zeroed out, then we refer to that controller with only the letters of the terms that are used. More specifically, there are controllers which use

- *PD control*, where $T_i = \infty$,
- *PI control*, where $T_d = 0$, and
- *P control*, where both $T_i = \infty$ and $T_d = 0$.

In the case of P control, there will always be a *steady-state error*, which is defined as $\lim_{t\to\infty} e(t)$, the difference between the desired output and the measured output in the limit as time goes to infinity. Incorporating integral action (by decreasing the integral time T_i) leads to a reduction of the steady-state error, but it also leads to an undesirable increase in the system's tendency to oscillate. We can counter this by including the derivative term, which adds damping to the system as T_d increases until reaching a certain threshold determined by the dynamics of the system.

6.3 Path-Following Control

Whereas some controllers are designed to keep a system at a stationary setpoint, other kinds of controllers drive a system along a time-varying trajectory. An interesting and useful application of trajectory-following control in robotics is to mobile robots. Since many mobile robots have nonholonomic constraints, the control problem becomes challenging due to the fact that error corrections must comply with motion constraints.

The route to be followed by a mobile robot can be expressed as a trajectory parameterized by time,

$$p_t(t) = \begin{bmatrix} x_t(t) & y_t(t) & \theta_t(t) \end{bmatrix}^T, \tag{6.5}$$

or a path parameterized by length,

$$p_s(s) = \begin{bmatrix} x_s(s) & y_s(s) & \theta_s(s) \end{bmatrix}^T. \tag{6.6}$$

Translating between these functions is a matter of reparameterizing via a function $r\colon [0, t_f] \to [0, s_f]$, so that $p_t(t) = p_s(r(t))$.

6.3.1 Path-Following Error Types

Suppose we have a target trajectory for a mobile robot to follow. If we define frame $\{t\}$ at the desired instantaneous pose $p_t(t)$, then the actual robot pose p_B^t represents the deviation of the robot's state from the desired state. The deviation is described in terms of three errors that correspond to the three rigid-body freedoms. All three errors are interrelated since an attempt to correct one has the potential to increase the others. Fig. 6.2 shows an illustration of the three error types, which are as follows:

- The **along-track error**, δs, is the distance ahead or behind the target in the instantaneous direction of motion. We write along-track error as $\delta s = -x_B^t$. Intuitively, along-track error can be reduced simply by adjusting the robot's velocity.

- The **cross-track error**, δn, is the portion of the position error orthogonal to the intended direction of motion. We write cross-track error as $\delta n = -y_B^t$. Cross-track error is harder to correct since it requires a velocity in a direction that is forbidden by the nonholonomic constraint, if one exists.

- The **heading error**, $\delta\theta$, is the difference between the desired heading and the actual heading. We write heading error as $\delta\theta = \theta_t - \theta_B$.

Each of the three path-following errors matches the form we saw in Equation 6.2, $e(t) = r(t) - y(t)$.

6.3.2 Open-Loop Path Following

An open-loop path-following controller executes the command that would be required to follow the path if the mobile robot were located at the correct pose $p_t(t)$ at

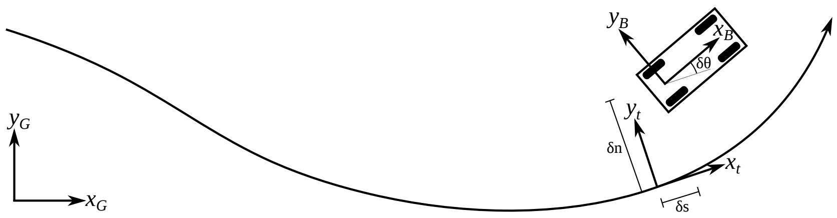


Figure 6.2: The three types of path-following error are cross-track error (δn), along-track error (δs), and heading error ($\delta\theta$). The three errors are interrelated since attempts to control one variable often affect the others.

time t. We can write an open-loop controller for a robot that is naturally controlled via angular velocity, such as a differential-drive robot:

$$u_{\omega,OL}(t) = \begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = \begin{bmatrix} \sqrt{\dot{x}_t(t)^2 + \dot{y}_t(t)^2} \\ \dot{\theta}_t(t) \end{bmatrix}. \tag{6.7}$$

We can similarly control a mobile robot with car-like steering via curvature:

$$u_{\kappa,OL}(t) = \begin{bmatrix} v(t) \\ \kappa(t) \end{bmatrix} = \begin{bmatrix} \sqrt{\dot{x}_t(t)^2 + \dot{y}_t(t)^2} \\ \frac{\dot{\theta}_t(t)}{\sqrt{\dot{x}_t(t)^2 + \dot{y}_t(t)^2}} \end{bmatrix}. \tag{6.8}$$

Naturally, these open-loop controllers will not correct any errors that occur during execution due to wheel-slip or to models that map linear and angular velocity into low-level actuator control inputs. We can therefore expect that error increases over time proportional to the square root of distance traveled. In particular, a heading error will tend to amplify a cross-track error over time.

6.3.3 Pure Pursuit: PD Control Method

To implement feedback control for path-following on a mobile robot, we can use the *pure pursuit algorithm*. It is defined based on a correction to the open-loop path-following controller, as

$$\begin{aligned} u_{\kappa,CL}(t) &= u_{\kappa,OL}(t) + \delta u(t) \qquad (6.9) \\ &= u_{\kappa,OL}(t) + K \begin{bmatrix} \delta s \\ \delta n \\ \delta\theta \end{bmatrix}, \qquad (6.10) \end{aligned}$$

where K is a gain matrix that is used to tune the controller. Let

$$K = \begin{bmatrix} k_s & 0 & 0 \\ 0 & k_n & k_\theta \end{bmatrix}, \tag{6.11}$$

where each of the k terms is non-negative. The pure-pursuit controller produced by this gain matrix performs a hybrid of P- and PD-control. It uses a simple P-controller to correct along-track error by simply speeding up or slowing down the robot. The control on curvature κ is a PD-controller for cross-track error because $\delta\theta$ is related to the derivative of δn.

The gain k_θ tends to decrease the rate of reduction in cross-track error, which is why it is a damping term. If we zero out k_θ, we get a P-controller on κ, which will tend to oscillate left and right along the path. Correcting the cross-track error generally induces a heading error, so that by the time cross-track error becomes zero, the heading is off and the robot misses the path entirely.

In general, corrections on cross-track error and heading error tend to counteract one another. As the robot moves closer to the path, cross-track error shrinks, and the heading error dominates more. As a consequence, the controller will begin to drive the robot more parallel to the path. The emergent effect of the pure pursuit controller is that at any moment in time, it seeks to reacquire the path smoothly at some future point in time and at a forward position along the path.

6.3.4 Pure Pursuit: Geometric Method

A geometric intuition about the pure pursuit controller (depicted in Fig. 6.3) may be helpful for its implementation. This model uses *receding horizon control*, in which the controller maintains an explicit target at some fixed lookahead time (or distance) along the path. As the robot moves, the target recedes so that its relative interval of time (or distance) along the path remains constant.

At each iteration of the controller, the robot computes a corrective path that will intersect the desired path at the horizon target. The form used for this corrective path is a constant-curvature arc, which is uniquely specified by the robot's current position and heading (because it can instantaneously move only straight ahead) as well as the final target point. The curvature of that arc may change with each iteration to reflect the shape of curve necessary to intersect the path at the horizon.

With each iteration, the robot executes a constant velocity and curvature for a short duration of time. The subsequent changes in curvature lead to a nearly smooth traversal that reacquires the path.

We can compute the required instantaneous curvature. Consider the geometric layout of the problem, as depicted in Fig. 6.4. We fit a constant-curvature arc such that it is tangent to the current position and orientation of the robot and intersects the horizon point on the path. By computing the coordinates of the horizon in the

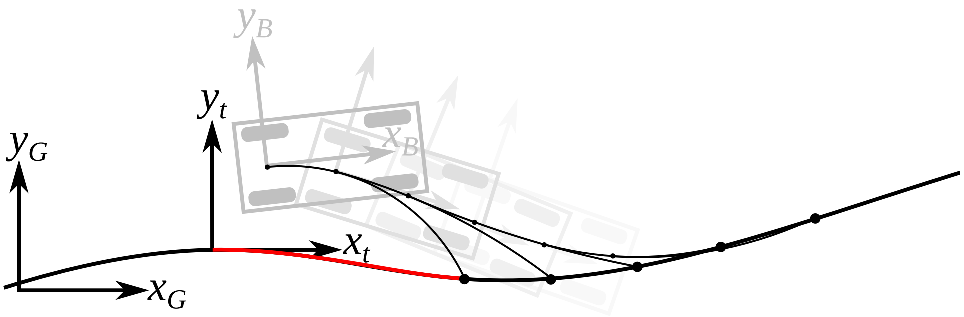


Figure 6.3: A pure pursuit controller can be implemented using a receding horizon (initially the red segment). Under this implementation, a constant-curvature arc is fitted between the current pose of the robot and the horizon point on the path. Each arc connects a small dot at the robot's current pose to a large dot at the current horizon.

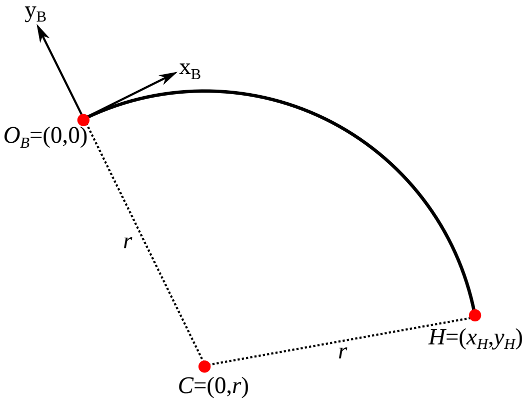


Figure 6.4: To compute the curvature of the arc that is needed to reacquire the path at the (receding) horizon, find a circle that intersects both the current position of the robot and the horizon point on the path, and which is tangent to the current direction of motion of the robot. As shown, the figure depicts a "negative" radius because clockwise motion decreases heading and is thus a negative curvature.

body frame (x_H^B, y_H^B), we can solve for the radius of that path as

$$\begin{aligned} r^2 &= (x_H^B)^2 + (r - y_H^B)^2 \\ &= (x_H^B)^2 + r^2 - 2ry_H^B + (y_H^B)^2 \\ 2ry_H^B &= (x_H^B)^2 + (y_H^B)^2 \\ r &= \frac{(x_H^B)^2 + (y_H^B)^2}{2y_H^B}. \end{aligned}$$

Finally, because the radius is the reciprocal of curvature,

$$\kappa = \frac{2y_H^B}{(x_H^B)^2 + (y_H^B)^2}.$$

By recomputing and executing this curvature in a loop, the robot will tend to smoothly approach and follow the path. Note that unlike the PD method of pure pursuit, geometric pure pursuit replaces the open-loop controller rather than adding an additional term to it.

6.3.5 Further Reading

For more information on path-following algorithms, see Kelly [2].

6.4 Controller Performance and Tuning

To make a controller perform as desired, we need to tune its various parameters, which we often call *gains* when they do not have a physical meaning. Tuning these parameters appropriately requires understanding the impact that each parameter has on the performance of the controller. In this section, we will see how controller performance affects system performance and then discuss methods for tuning PID controllers and path-following controllers to achieve desirable behaviors.

6.4.1 Characterizing Performance

For any given system, there is not a single correct way to tune its controller. Rather, the tuning choices represent a set of tradeoffs, which manifest themselves in how the controller performs. We will highlight several of these tradeoffs below.

Convergence to the Setpoint

One tradeoff of control systems is the *damping ratio*, denoted by ζ, which measures the performance of the controller combined alongside the system being controlled.

The damping ratio influences how and whether the system will reach the setpoint. We consider four cases below.

- If $\zeta = 0$, then the system is **undamped** and will oscillate forever. Since there is no damping, such a system corresponds to P (pure proportional) control with a large K_p term.
- If $0 < \zeta < 1$, then the system is **underdamped** and will oscillate for many cycles, eventually stabilizing to a steady state. In the case of PID control, an underdamped system corresponds to a PD controller in which the K_d term is too small relative to the K_p term.
- If $\zeta > 1$, then the system is **overdamped** and will approach the setpoint very slowly but without any oscillation. In the case of PID control, an overdamped system corresponds to a PD controller in which the K_d term is too large relative to the K_p term.
- If $\zeta = 1$, then the system is **critically damped** and will quickly converge to the setpoint without oscillating and remain there. We can model this situation using a PID controller where the K_p and K_d terms are appropriately chosen with respect to one another.

Resistance to Setpoint Deviation

Another important tradeoff to consider is how resistant the system is to deviation from the setpoint. We distinguish between two types of systems: those that strongly resist deviation, and those that do not.

- A **stiff** system resists deviation from the setpoint and can be characterized by a high K_p term. When a stiff system is at the setpoint, it takes a great deal of effort to remove it from the setpoint. If sufficient effort is applied, the system may oscillate at a high frequency (higher than the characteristic frequency of the uncontrolled system). Stiff systems are preferred when the system needs to be very precise (such as an industrial manipulator arm that performs welding tasks) because stiffness resists disturbances that would reduce precision. The major disadvantage of a stiff system is that it is likely to cause harm or damage if it comes into contact with an object.
- A **compliant** system is much more tolerant of not being at the setpoint and can be characterized by a small K_p term. If allowed to oscillate, the system will do so close to its characteristic or resonant frequency. Compliant systems are generally preferred in situations where contact is involved, in

particular contact with humans. In addition, compliance is useful in situations such as grasping an object whose precise geometry is unknown; the hand can attempt to grasp a smaller target inside of the true object and rely on compliance to gently squeeze the object without causing damage. However, since they deviate from the setpoint without much effort, compliant systems often suffer from imprecision.

Ability to Overcome Forces

When tuning a controller, we need to consider what external forces will affect the system being controlled. For example, suppose that the system is a manipulator arm that needs to slide parts against each other in order to perform assembly tasks. If there is high static friction between the parts, then it likely will require more force to begin sliding one of the parts from a resting position than to continue sliding it once it is moving. In this case, we may choose to use a higher integral gain since integral control can build up over time and help get the part moving. In contrast, if both the static friction and sliding friction are high, the system would be better controlled by increasing the proportional and derivative gains. In general, integral control is considered somewhat dangerous due to the fact that it can build up and spontaneously release a lot of energy into the system being controlled. For many applications, it is better to minimize or avoid the use of the integral gain for safety.

6.4.2 Tuning PID Controllers

There are a variety of methods used for tuning PID controllers, and no one method is guaranteed to produce an optimal tuning for all controlled systems. But for many applications in robots, a good starting point is the **Ziegler-Nichols tuning method**, which involves the following steps:

1. Set all gains to zero.
2. Increase the K_p gain a small amount.
3. Wait for the system to restabilize.
4. Repeat steps 2–3 until the system has stable and consistent oscillations.
5. Define a new gain K_u, the *ultimate gain*, and set it equal to the current K_p.
6. Measure the current oscillation period and call it T_u.
7. Set $K_p = 0.6K_u$.
8. Set $T_i = T_u/2$.
9. Set $T_d = T_u/8$.

The constant factors in steps 7–9 seem arbitrary, but they were arrived at through experimentation. Ziegler and Nichols hand-tuned PID controllers for a variety of systems until they achieved the desired performance. They then looked for a small set of parameters that would explain the patterns they saw, which were K_u and T_u.

In hand-tuning their PID controllers, the desired performance that Ziegler and Nichols sought was a *quarter amplitude decay ratio*. What this means is that the controller is designed to be underdamped — it overshoots the setpoint several times before converging — but the amplitude of each overshoot is one-quarter of the previous overshoot. As previously stated, this tuning is not guaranteed to be optimal, but it is often a satisfactory initial estimate of a more optimal tuning.

6.4.3 Tuning Path-Following Controllers

Recall from Section 6.3 that the pure-pursuit algorithm for path-following can be implemented either with PD control or geometrically with receding horizon control. Since these two methods have a different set of parameters, they have different tuning procedures, which we describe below.

Tuning PD Pure Pursuit

The PD-control form of pure pursuit can be tuned like any PD controller according to the Ziegler-Nichols method. During tuning, the robot should follow a long, straight-line, constant-velocity path. That way, the target frame moves in an orderly and predictable fashion along the path. Consequently, any changes in the three errors are induced by the behavior of the controller alone.

The first step is to tune the PD controller for cross-track error. To do this, we can use the Ziegler-Nichols method while ignoring the I-gain K_i. We let $K_p = k_n$ and $K_d = k_\theta$ and then set those two gains according to the Ziegler-Nichols method.

The second step is to tune a P controller for the along-track error. We again can employ the basic Ziegler-Nichols method while ignoring both the I-gain and the D-gain. We let $K_p = k_s$ and then use Ziegler-Nichols to set the gain.

Since the Ziegler-Nichols method as described in these notes intends for you to be tuning a full PID controller, omitting one or two gains may result in suboptimal performance. Variations to the method have been proposed that suggest alternate tuning formulas, but we do not cover those here. After tuning with Ziegler-Nichols, one can fine-tune the controller's performance by hand, although doing this is most intuitive with simple controllers like P-only and PD controllers.

After the straight-path tuning is complete, the resulting controller should also work on curvy paths due to its inclusion of the open-loop path-following term u_{OL}.

Tuning Geometric Pure Pursuit

In contrast to the PD-control method, the geometric method of pure pursuit is quite simplistic because there is only one parameter to tune: h, the lookahead distance of the horizon along the path. The downside of its simplicity is that it may not follow a path as well as the PD-control method. The reason for this is that driving the robot directly towards the lookahead horizon has the effect of ignoring fine details that occur earlier than the horizon point. Therefore, geometric pure pursuit tends to smooth out the tightest curves in the path.

The choice of lookahead h governs the responsiveness of the robot to path-following error and represents a tradeoff, which is summarized below.

- If h is too **small**, then the robot will exhibit **underdamped** behavior: it will quickly cancel out errors by turning sharply toward the path, possibly overshooting. If the robot encounters a sharp corner in the target path, it will tend to overshoot the corner, especially if it has steering angle limits. Overall, the robot will try to reproduce the shape of the path with high precision.

- If h is too **large**, then the robot will exhibit **overdamped** behavior: it will be less responsive to error and will approximate the shape of the target path more coarsely. If the robot encounters a sharp corner in the target path, it will tend to cut the corner on the inside. Overall, the robot will reproduce the shape of the path with lower precision. It will follow a smoothed version of the path.

6.5 Model-Predictive Control

In Section 6.3, we learned that pure pursuit can be implemented geometrically with *receding horizon control*, where at each timestep the robot computes a curvature that will move the robot from its current position towards a horizon point receding along the desired path of the robot. In this section, we will learn about another type of receding horizon control called **model-predictive control** (MPC).

Model-predictive control is a numerical method for finding optimal control signals parameterized in time. The “model” in model-predictive control is a numerical simulation, which can incorporate both dynamics and kinematics, that is designed to closely mimic the behavior of the real system being controlled. This model serves as a proxy for the real system being controlled; we assume that if the model predicts that a certain control signal is effective in simulation, then it will be effective on the real robot too.

Model-predictive control operates three nested, iterative loops, one inside the other (Fig. 6.5). At the outer layer, we have receding horizon control, which iteratively computes a path to a point some distance into the future. Within each loop of receding horizon control, MPC solves an optimization problem via a numerical method that iterates over different possible control signals to find the lowest cost. Finally, within each step of the numerical optimizer, the innermost loop numerically integrates the control signal through time to produce a candidate path.

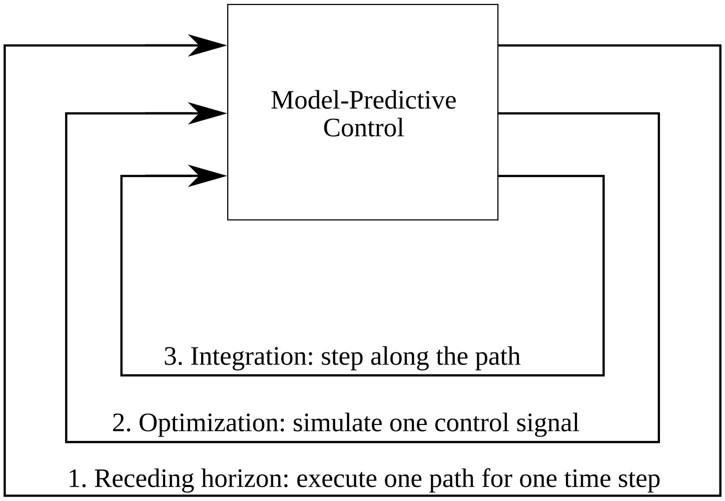

Model-Predictive Control
3. Integration: step along the path
2. Optimization: simulate one control signal
1. Receding horizon: execute one path for one time step

Figure 6.5: The three nested loops of model-predictive control.

At the outermost loop, model-predictive control is a type of receding horizon control because at each timestep it computes a control plan for a short future time interval that “recedes” with time. In other words, at time t_i, the controller plans a control strategy for the time interval $[t_i, t_i + \Delta t]$, where Δt is a fixed time horizon. After selecting and executing a control signal for time t_i, MPC then begins to compute a control strategy for the partially-overlapping interval $[t_{i+1}, t_{i+1} + \Delta t]$, as shown in Fig. 6.6. When iteration terminates at the outer loop, it means the robot has reached its goal.

The middle loop is solving a numerical optimization problem by searching over the space of paths. The path shape is evaluated by an objective function for comparison to other candidate paths. When iteration terminates at the middle level, it means that the allotted computation time Δt has expired. Since the algorithm runs in real time while the robot is moving, it has an upper bound on the amount of time it can spend exploring alternative paths.

In the innermost loop, the simulator takes as inputs a state and a control input and integrates the state of the system some time interval in the future. Most often, this integral is solved by an iterative approach such as Euler integration. It is this lowest level that actually predicts where a particular control would lead. When the

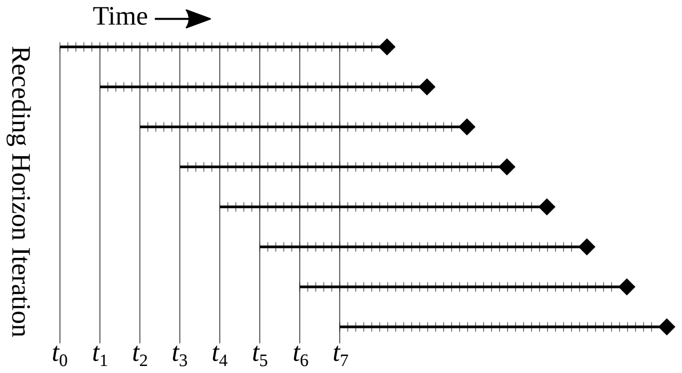


Figure 6.6: This figure depicts time, both simulated and executed on the real robot. The time steps $t_0, t_1, \ldots$ denote time steps of receding horizon control. At a finer-grained simulation step size, the predictive model looks ahead much deeper than the interval that is actually executed on the robot. The lookahead enables the controller to have some confidence that it is headed towards the goal.

inner loop terminates, it means that the integral has reached the time horizon. At this point, the cost of the path is computed and returned to the next higher level. Costs that the controller could consider might include the average deviation of the robot's state from the desired state, average motor torque, total energy required, and the final position and orientation of the robot when reaching the horizon (since these are predictors of future performance beyond the horizon time).

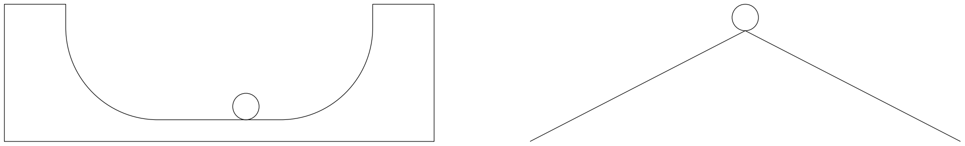
Since MPC recomputes a control signal and executes it on the robot once per Δt seconds, the control designer must decide how best to utilize the time allotment. There is a fundamental tradeoff between the depth into the future, measured in time horizon seconds, and the breadth, measured by the number of paths evaluated. Halving the horizon time allows the controller to explore twice as many paths. Another tradeoff involves the fidelity of the model. Higher fidelity models generally take longer to compute, both because they perform more sophisticated computations and also because they may more finely discretize the time when performing integrals, thus necessitating more integration steps. Therefore, if the fidelity of the model doubles, then either the lookahead distance or the number of paths evaluated must be halved. A final tradeoff involves the choice of Δt. Increasing Δt permits a longer computation time, which can be used to increase model fidelity, number of paths evaluated, or the depth of the paths. However, a larger Δt also increases the elapsed time before the robot reacts to changes in its environment. Therefore, a reasonably small Δt such as 0.1 sec is typically desired for real-time, reactive control behavior.

6.6 Stability

When designing a controller for a dynamical system, it is important that we verify whether the controller makes the system **stable** or not. There are many different types of stability within control theory, each with their own technical definition, but the details of all these types are out of the scope of this section. We will instead discuss what stability is at a high level, why it matters, and then show one method for formally describing and analyzing the stability of a system.

An intuitive way to think about stability is that it measures the tendency that the response of a dynamical system will return to an *equilibrium point* after the system is disturbed. If a system is at an equilibrium point, then its dynamics are in a stationary condition, which is one where all forces balance. More formally, we describe the system dynamics by $\dot{V}(x)$, which captures the state of how the system evolves in time. Then a state x_e is an equilibrium point of the system if $\dot{V}(x_e) = 0$. In general, dynamical systems may have zero, one, or more equilibrium points.

We will illustrate these concepts with the simple systems shown in Fig. 6.7. Suppose that the ball shown in Fig. 6.7a is disturbed by a force. Provided that the force does not knock it out of the half-pipe, and assuming that there is friction, the ball will eventually come to rest at one of the many equilibrium points along the flat bottom of the half-pipe. Now suppose that the ball shown in Fig. 6.7b is disturbed by a force. It will roll down one of the two sides of the mountain, away from the single equilibrium point at the peak, and if we imagine that the sides of the mountain continue indefinitely, then the ball will roll indefinitely.



(a) A system in a stable equilibrium.

(b) A system in an unstable equilibrium.

Figure 6.7: The system in (a) is resistant to disturbances, whereas the system in (b) is not.

6.6.1 Lyapunov Stability

Knowing the extent to which a system is stable is crucial because an unstable system has the potential to be dangerous. For example, an unstable mobile robot might tip over and crash, or a manipulator arm might swing around unpredictably. These risks motivate the need for a formal definition and method to test the stability of a system. In these notes, we will use *Lyapunov stability*, which looks at how a system behaves near a point of equilibrium. Before giving the technical definition of Lyapunov stability, we will first provide some intuition.

If you have been to a science museum, there is a good chance you have seen a coin vortex funnel, also called a "spiral wishing well". When you put in a coin, it rolls in a spiral orbit and slowly descends into the bottom of the funnel. The bottom of the funnel, where the coin eventually stops, is an equilibrium point. Let h be the height of the coin from the bottom of the funnel, and define $V(h) = h_0 - h$, so that $V(h)$ is everywhere positive except at the bottom. The function $P = mgh$ gives the potential energy of the coin, and the function $K = 0.5mv^2$ gives the kinetic energy[1]. The total energy of the coin is therefore given by $P + K$. When the coin is at its initial height, $K = 0$ because the coin has no velocity. As the coin starts to roll, K increases, meaning that P must decrease due to conservation of energy. Thus, h can never be greater than h_0, which implies that the coin can never escape the funnel. We can therefore conclude that the coin is a stable system.

The coin vortex funnel and the other examples are physical systems in which stability is measured by a balance of forces, such that the change in energy of the system is non-positive. We saw how the function describing the energy of the system as a function of state can be used to classify the stability of the system.

We can apply the same concept to any controlled system, even if energy does not govern stability. Suppose we have an arbitrary dynamical system governed by a differential equation $\dot{x} = f(x)$, where $x(t; a)$ is a solution starting from initial condition a at time t. Abstractly, we say that the system described by the differential equation is stable if there exists a neighborhood around a described by a ball of radius δ such that for any initial condition b within the ball, the difference between all future states from a and b is constrained by an upper bound ϵ. Formally,

$$\|b - a\| \leq \delta \implies \|x(t; b) - x(t; a)\| < \epsilon \text{ for all } t > 0. \tag{6.12}$$

This type of stability is referred to as *stability in the sense of Lyapunov*. An interpretation of this definition is that by starting sufficiently close to a certain initial condition, we are guaranteed to stay close to its corresponding solution. If a solution is stable in the sense of Lyapunov but trajectories starting from different initial conditions do not converge, then the solution is called *neutrally stable*. The half-pipe in Fig. 6.7a is neutrally stable since the bottom is flat and the ball does not always converge to a single point.

A solution $x(t; a)$ is called *asymptotically stable* if it is stable in the sense of Lyapunov and any solution in the neighborhood of attractor a converges to a. Formally,

$$x(t; b) \to x(t; a)\ as\ t \to \infty \text{ for b sufficiently close to } a. \tag{6.13}$$

[1] m is the mass of the coin, g is the acceleration due to gravity, and v is the velocity of the coin.

This is the case where all nearby trajectories converge to the stable solution as time goes to infinity. Specifically for planar systems, asymptotically stable equilibrium points are commonly referred to as *attractors*. In the vortex funnel example, the coin is asymptotically stable, and the bottom of the funnel is an attractor.

A solution is *locally stable* or *locally asymptotically stable* if it is stable for all initial conditions $x \in B_r(a)$, where $B_r(a) = \{x : \|x - a\| < r\}$ is a ball of radius r around a and $r > 0$. A system is *globally stable* if it is stable for all $r > 0$.

A solution $x(t; a)$ is *unstable* if it is not stable. Formally, a solution $x(t; a)$ is unstable if given some $\epsilon > 0$, there does not exist a $\delta > 0$ such that if $\|b - a\| < \delta$, then $\|x(t; b) - x(t; a)\| < \epsilon$ for all t. An unstable equilibrium point of a planar system is typically referred to as a *source*, if all trajectories move away from the equilibrium point, or *saddle*, if some trajectories lead to the equilibrium point and others move away. The mountain in Fig. 6.7b is an unstable system.

6.6.2 Lyapunov Stability Analysis

We are interested in proving that a given solution is stable, asymptotically stable, or unstable. Lyapunov Stability Analysis provides a formal tool for doing so, based on the definition of energy-like functions called the *Lyapunov functions*. For a given system, you can pick any function you want, provided it has the right properties.

A Lyapunov function is a nonnegative function $V : \mathbb{R}^n \rightarrow \mathbb{R}^+$ that always decreases along system trajectories; thus its minimum is a locally stable equilibrium point. Therefore, if we find a function with those properties for a given system, we can prove that the system is stable. Such a function is called a *Lyapunov function*. To characterize the stability of a system in the sense of Lyapunov, we make use of the *Lyapunov Stability Theorem*. To formulate the theorem, we first need to go over a few definitions:

- A continuous function V is *positive definite* if $V(x) > 0$ for all $x \neq 0$ and $V(0) = 0$.

- A continuous function V is *negative definite* if $V(x) < 0$ for all $x \neq 0$ and $V(0) = 0$.

- A continuous function V is *positive semidefinite* if $V(x) \geq 0$ for all x, but $V(x)$ can be zero at points other than just $x = 0$.

Lyapunov Stability Theorem

Let $V : \mathbb{R}^n \to \mathbb{R}^+$ and denote by $\dot{V}$ its time derivative along trajectories of system dynamics, as

$$\dot{V} = \frac{\partial V}{\partial x}\frac{dx}{dt} = \frac{\partial V}{\partial x} f(x). \tag{6.14}$$

Let also $B_r = B_r(0)$ be a ball of radius r around the origin. If there exists $r > 0$ such that V is positive definite and $\dot{V}$ is negative semidefinite for all $x \in B_r$, then $x = 0$ is *locally stable in the sense of Lyapunov*. If V is positive definite and $\dot{V}$ is negative definite in B_r, then $x = 0$ is *locally asymptotically stable*.

Example 6.1 Stability analysis of a mechanical system

Consider the system of Fig. 6.8 with a block of mass m attached to a spring of stiffness k. The block is located at position $x = 0$ and is subject to frictional force of coefficient b proportional to the block's velocity. We can write the motion equation as

$$m\ddot{x} + b\dot{x} + kx = 0 \tag{6.15}$$

and compute the total energy of the system as

$$V = \frac{1}{2}m\dot{x}^2 + \frac{1}{2}kx^2. \tag{6.16}$$

The rate of change of the total energy of the system can then be found by differentiating (6.16) with respect to time, as

$$\dot{V} = m\dot{x}\ddot{x} + kx\dot{x}. \tag{6.17}$$

Substituting (6.15) for $m\ddot{x}$ in (6.17), we obtain

$$\dot{V} = -b\dot{x}^2, \tag{6.18}$$

which is always nonpositive since $b > 0$. Therefore, energy always leaves the system unless $\dot{x} = 0$, which implies that the system will release the energy until it comes to rest regardless of initial state. Hence, we have shown that this spring-mass system will eventually come to rest at the equilibrium.

Further Reading

For more on stability, please see Sections 4.3 and 4.4 of Åström and Murray [1].

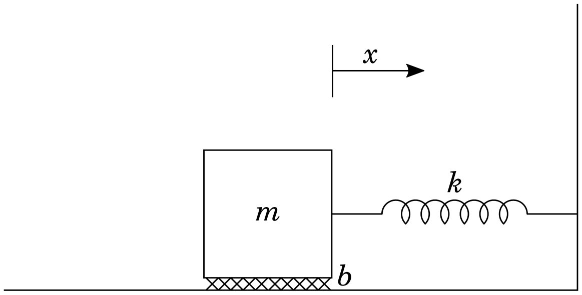


Figure 6.8: Simple spring-mass system subject to friction.

Bibliography

[1] K. J. Åström and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2010.

[2] A. Kelly. *Mobile Robotics: Mathematics, Models, and Methods*. Cambridge University Press, 2013.