RBE 2002: Unified Robotics II
Lab 2: Motor and motion control

Last modification: October 28, 2022

Motion control is a core function of almost every robot. For many robots, the primary focus is on position control, for example a welding robot in an industrial setting or a surgical robot. For others, velocity control is most important, for example, with autonomous vehicles.

In each case, the overall control of the robot typically has a hierarchical structure: the desired motion or position of the robot is first calculated and then the robot controls motors to execute the desired action.

In this lab, you'll start by developing the "low-level" speed controller for your Romi and then explore higher-level control, in this case, wall following.

In modern robots, there is often a hierarchical architecture to the system, with multiple processors focusing on different tasks. When dividing up functionality across multiple processors, a "typical" architecture will have a dedicated, low-level processor (like the `ATmega32U4`) to manage the velocity control of motors and a higher level processor (like an NVIDIA Jetson or Raspberry Pi or more powerful computer) to calculate a robot trajectory, plan a path, and so forth. The high-level processor would then send motor commands to the lower level controller. Such an architecture is exactly what is used in RBE 3001 and RBE 3002 (in those cases, the high-level processing is done on a computer running `MATLAB` or `ROS` and the low-level control on a SAMD51 or Dynamixel).

In this lab, even though all of your code runs on one processor, you will still develop a hierarchical controller for controlling the motion of your Romi. In this case, you will build a PI controller for motor speed and then use it for wall following with inputs from your sensors.

Control strategies

How you define the controller affects the performance of a robot. A relatively simple controller, where all the dynamics are condensed into one block, may be easier to code, but it may underperform with respect to a more complex controller. In addition to better performance, more complex controllers are often more flexible, as they allow the designer to address different components independently (think modular).

Consider, for example, a line-following controller. One way to accomplish line following is to "connect" the output of the line sensor(s) directly to the motor effort (the voltage supplied to the motor). Depicted in Figure 1, the procedure starts with determining the deviation from the line. That deviation is fed to a controller (typically a PID controller) that directly adjusts the voltages going to the motors. This method works reasonably well and is very easy to code:

```
lineError = CalcErrorFromLineSensors();
turnEffort = CalcEffortPID(lineError);
```

Figure 1: A single step algorithm for line following.

```
rightMotor.SetEffort(baseEffort + turnEffort);
leftMotor.SetEffort(baseEffort - turnEffort);
```

where `baseEffort` is the effort (voltage) needed to drive at some nominal speed. The remaining commands should be self-explanatory.

Another way to execute line-following control is to use a two-step procedure, as depicted in Figure 2. In the first step, a line-tracking controller calculates the wheel *velocities* needed to correct the vehicle's motion. These *target* velocities are then fed into controllers for each wheel, which adjust the effort (voltage) to their respective motor, based on the error between the target and actual wheel speeds.

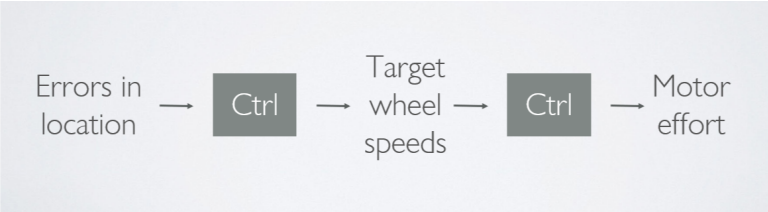

Figure 2: A two step procedure for line following.

The advantages of this formulation are that the control system can use different controllers at each stage, and it's easier to account for variations in individual motors. It's a little more difficult to code and tune, but compartmentalizing the controller functions can make it perform better:

```
lineError = CalcErrorFromLineSensors();
turnSpeed = CalcEffortPID(lineError);

rightMotor.SetTargetSpeed(baseSpeed + turnSpeed);
rightError = rightMotor.CalcError();
rightEffort = CalcMotorEffortPID(rightError);
rightMotor.SetEffort(rightEffort);

leftMotor.SetTargetSpeed(baseSpeed - turnSpeed);
.
.
.
```

Figure 3: An open-loop control algorithm.

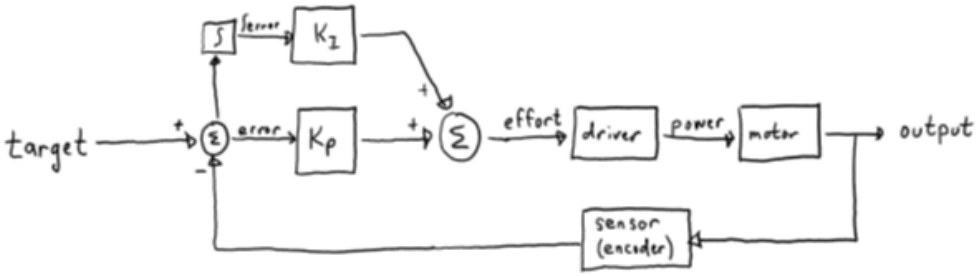

Figure 4: A PI controller.

Open-loop and feedforward control

When you first explored PID control, you probably saw the "classic" cruise control problem: If you were driving a car along a flat road, you could derive a relationship between throttle (gas pedal) position and speed. So long as conditions remained the same (flat road, no wind, etc.), you could blindly[1] use this relationship to drive at the speed you wanted. Depicted in Figure 3, Such control is generally called *open-loop control*:

```
effort = CalcEffort(targetSpeed); //maybe a simple, linear function
throttle.SetEffort(effort);
```

Of course, as soon as there are any disturbances (hills, wind, etc.), the relationship breaks down, which motivates the use of a *feedback controller*, such as a PID controller (see Figure 4):

```
error = targetSpeed - actualSpeed;
effort = CalcPID(error);
throttle.SetEffort(effort);
```

In closed-loop control, the speed of the motor is measured by a sensor – e.g., the Romi's encoders – and the error is fed back to the controller, as shown in Figure 4. In this example, there is a proportional term and an integral term. As discussed in class, the integral control term is used to remove the *offset* or *steady-state error*.

On a side note, there's no reason that we need to "throw out" the open-loop model when creating a PID controller. Even if the open-loop control is inaccurate, so long as there is a feedback controller

[1] Speaking of blind, have we talked about Massachusetts drivers yet?

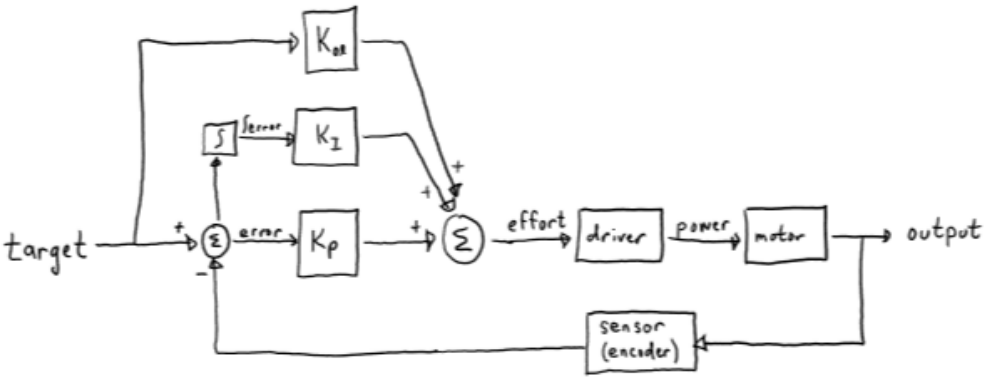

Figure 5: A PI controller with an open-loop term.

layered on top of it, we can still guarantee that we can reach the target speed. The advantage of doing so is that we can get a faster response to changes in target speed: the open-loop contribution is immediate, whereas a PID controller has to wait until the integral term builds up enough to compensate for changes in the target speed. Figure 5 shows a combined controller.

Feedforward control is similar to an open-loop control, in that the effort is directly fed to the plant. The difference is that a feedforward controller uses measurements of the *disturbances* to apply corrections, where as the open-loop term is based on a disturbance-free model. In our example above, if you had a way to measure the grade of a hill, you could augment your model to account for the extra effort needed to go up it.