Chapter 6

Robot Control

Most people who have seen past 6.270 contests (or any of the raw videotape footage of past contests), remember round after round of wedged robots—robots that got stuck against a wall, or stuck against another robot, and stayed that way for the duration of a round.

Often these robots did not suffer from mechanical or electronic failures, but rather from poor software design. Perhaps a robot were running into the wall, and its front touch sensor did not trigger. This robot was not physically stuck, but it was "mentally stuck": its program did not account for this situation, and did not provide a way for the robot to get free.

Robot after robot has failed in this way, infuriating their creators and boring the 6.270 audience. This chapter will discuss the "hard truth" about robot sensors and present a framework for thinking about control that will help the reader prevent his or her robot from suffering the same fate.

A few words of advice: most people severely underestimate the amount of time that is necesssary to write control software. A program can be hacked together in a couple nights, but if a robot is to be able to deal with a spectrum of situations in a capable way, more work will be required.

Also, it's very painful to be developing contest-ready software at the same time as one is making changes hardware. Any hardware change will necessitate software changes. Some of these changes may be obvious but others will not. One sad story is of a team that revamped their robot's entire geartrain in between the first and second nights of the contest. The new geartrain performed much better, but all of the timing parameters in their code needed to be adjusted. The team never recovered from their mechanical "upgrade."

In the past it has been observed that that the most successful robots were the ones that had their mechanics and sensors completed about a week before contest. We urge you to make this your goal: Finish your robot with a week to go, and develop your software based upon a stable hardware platform.

6.1 Basic Control Methods

6.1.1 Feedback Control

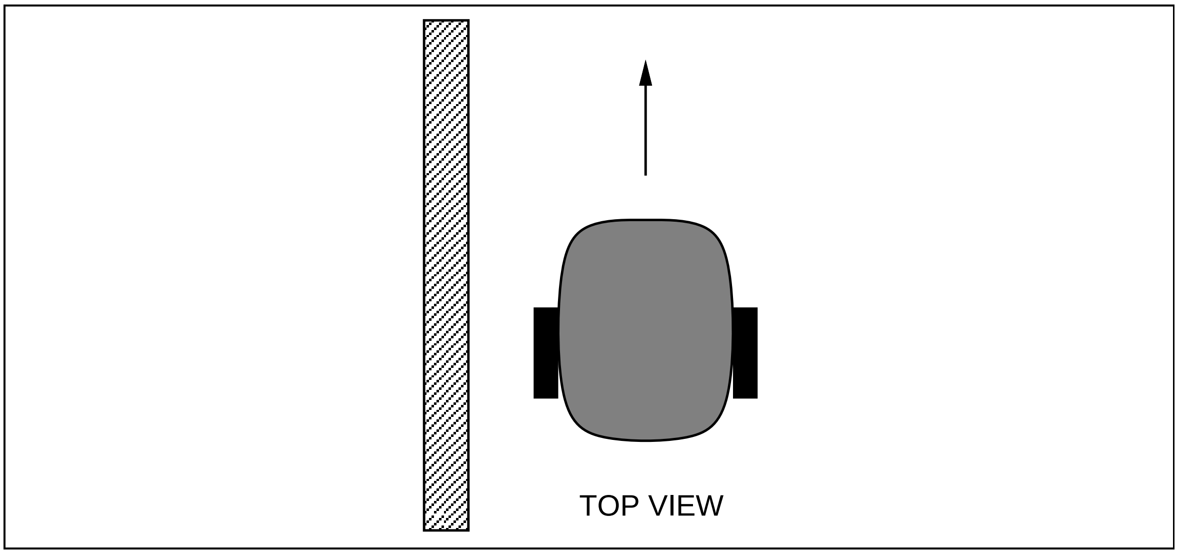

Figure 6.1: Driving along a Wall Edge

Suppose it is desired for the robot to drive with its left side is near the edge a wall, following the wall edge (see Figure 6.1). What's the best way to accomplish this activity?

One solution would be to orient the robot exactly parallel to the wall, and then drive straight ahead. However, this simple solution has two problems: if the robot is not initially oriented properly, it will fail. Also, unless the robot were extremely proficient at driving straight, it will eventually veer from its path and drive either into the wall or into the playing field.

The common and effective solution is to build a negative feedback loop. With continous monitoring and correction, a goal state (in this case, maintaining a constant distance from a wall) can be achieved.

The bend sensors provided in the 6.270 kit would make an excellent sensor to measure the distance from robot and the wall (see Figure 6.2). The closer the robot is to the wall, the more the bend sensor is bent.

Suppose a function were written using the bend sensor to discerned three states: `TOO_CLOSE`, `TOO_FAR`, and `JUST_RIGHT` (from the wall). Here is a possible definition of the function, called `wall_distance()`:

```
int TOO_CLOSE= -1;
int JUST_RIGHT= 0;
```

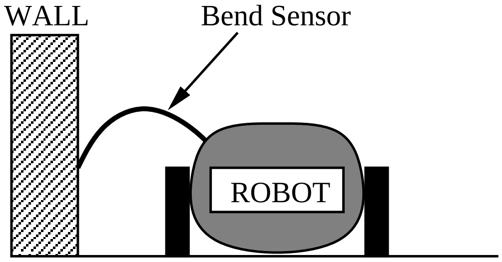

Figure 6.2: Using a Bend Sensor to Measure Distance to a Wall

```
int TOO_FAR= 1;

int wall_distance()
{

    /* get reading on bend sensor */
    int bend_value= analog(BEND_SENSOR);

    /* assume smaller values mean more bending of sensor */
    if (bend_value < TOO_CLOSE_THRESHOLD) return TOO_CLOSE;
    if (bend_value > TOO_FAR_THRESHOLD) return TOO_FAR;
    return JUST_RIGHT;

}
```

Now, a function to drive the robot making use of the `wall_distance()` function would create the feedback. In this example, the functions `veer_away_from_wall()`, `veer_toward_wall()`, and `drive_straight()` are used to actually move the robot, as shown in Figure 6.3.

```
void follow_wall()
{
    while (1) {
        int state= wall_distance();

        if (state == TOO_CLOSE) veer_away_from_wall();
        else if (state == TOO_FAR) veer_toward_wall();
        else drive_straight();
    }

}
```

Figure 6.3: Wall-Following Function

Even if the function to drive the robot straight were not exact (maybe one of the robot's wheels performs better than the other), this function should still accomplish its goal. Suppose the "drive straight" routine actually veered a bit toward the wall. Then after driving straight for a bit, the "follow wall" routine would notice that the robot was too close to the wall, and execute the "veer away" function.

The actual performance of this algorithm would be influenced by many things, including:

- How sharply the "veer away" and "veer toward" functions made the robot turn.
- How well the bend sensor measured the distance to the wall.
- The settings of the the `TOO_CLOSE_THRESHOLD` and `TOO_FAR_THRESHOLD` values.
- The rate at which the `follow_wall()` function made corrections to the robot's path.

Still, use of a negative feedback loop[1], ensures the construction basically stable and robust performance, once the parameters are tuned properly.

6.1.2 Open-Loop Control

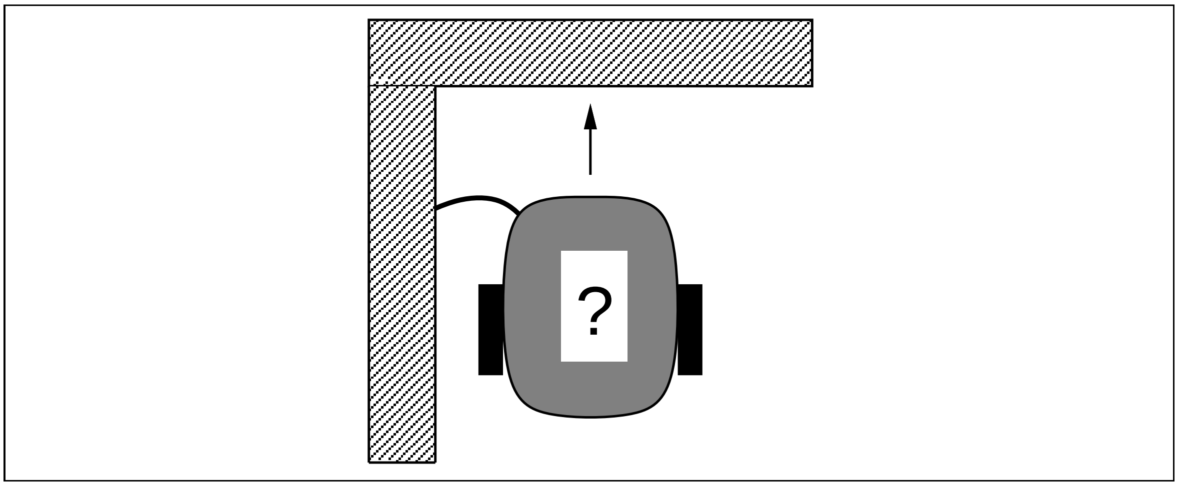

Figure 6.4: Negotiating a Corner

Suppose now the robot has been following the wall, and a touch sensor indicates that it has reached the far edge. The robot needs to turn clockwise ninety degrees to

[1]The type of feedback just described is called *negative* feedback because the corrections subtract from the error, making it smaller. In *positive* feedback, corrections add to the error. Such systems tend to be unstable.

continue following the edge of the wall (see Figure 6.4). How should it accomplish this?

One simple method would be to back up a little and execute a turn command that was timed to accomplish a ninety degree rotation. The following code fragment illustrates this idea:

```
....
robot_backward(); sleep(.25); /* go backward for 1/4 second */
robot_spin_clockwise(); sleep(1.5); /* 1.5 sec = 90 degrees */
....
```

This method will work reliably only when the robot is very predictable: if a turn command of 1.5 seconds always produces a rotation of 90 degrees. As might be guessed, this is far from being the case. Many factors affect the performance of a timed turn, including the battery strength, traction on the surface, and friction in the geartrain.

This method is called *open-loop control* (as compared to *closed-loop control*) because there is no feedback from the commanded action as to its effect on the state of the system. If the command is tuned properly and the system is very predicable, open-loop commands can work fine, but generally, closed-loop control is necessary for good performance.

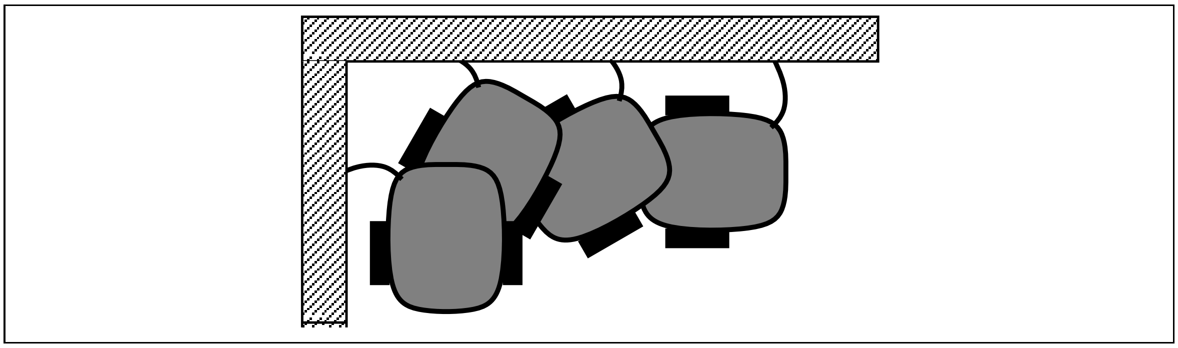

Figure 6.5: Negotiating a Corner with Touch Sensor Feedback

How could the corner-negotiation action be made into a closed-loop system? One approach (which has been used successfully in past 6.270 robots) is to have the robot make little turns, drive straight ahead, bonk the wall, back up, and repeat (see Figure 6.5). The corner is negotiated with a series of little steps.

6.1.3 Feed-Forward Control

There are certain advantages to open-loop control, most notably: speed. It's easy to see that a single timed turn would be much faster than a set of small turns, bonks, and back-ups.

One approach when using open-loop control is to use *feed-forward* control. This means that the commanded signal is a function of some parameters that are measured in advance. For the timed turn action, battery strength is probably one of the most significant factors determining the required time of the turn. Using feed-forward control, a battery strength measurement would be used to "predict" how much time is needed for the turn. Note that this is still open-loop control—the feedback is not based on the actual result of a movement command—but a computation is being performed to make the control more accurate.

For this example, the battery strength could be measured or estimated based on usage since the last charge.

6.1.4 Summary

For the types of activities commonly performed by 6.270 robots, feedback control is very useful for:

- **Wall following.** As discussed in this section.
- **Line following.** Using one or more reflectance sensors aimed at the surface of the 6.270 playing field.
- **Infrared tracking.** Homing in on a source of infrared light, using the Sharp IR sensors.

Open-loop control should probably be used only sparingly, and in time-critical applications. Ideally, small segments of open-loop actions interspersed between feedback activities should work well.

Feed-forward techniques can enhance the performance of open-loop control, when it is used.

6.2 Sensor Calibration

6.2.1 Manual Sensor Calibration

The function `wall_distance()` (used an example in the Section 6.1.1) used threshold variables (`TOO_FAR_THRESHOLD` and `TOO_CLOSE_THRESHOLD`) to interpret the data from the bend sensor. Depending on the actual reading from the bend sensors, and the settings of these threshold variables, `wall_distance()` determined if the robot was "too close," "too far," or "just the right" distance from the wall.

Proper calibration of these threshold values is necessary so that the robot can perform properly. Often, it is convenient to write a routine to allow interactive manipulation of the robot's sensors to determine the proper calibration settings.

For the bend sensor, a calibration routine could be included that allowed flexing the sensor for a small amount of bend (the `TOO_FAR_THRESHOLD`), and then depression of one of the user buttons. The routine would then "capture" the value of the bend sensor at that point, and use it as the appropriate threshold. Similarly, a large amount of flex could be applied to the sensor and then captured as the `TOO_CLOSE_THRESHOLD` value.

Later, the values of these thresholds could be noted when the robot is performing particularly well. These "optimal" settings could be hard-coded as as default values. The calibration routine could be kept, for use under circumstances, or if other parameters affecting the robot's performance cause the need for adjustment of the calibration settings again.

6.2.2 Dealing with Changing Environmental Conditions

Calibration routines are particularly important when environmental conditions will cause fluctuations in sensor values. Two types of sensor are strongly affected either by external environmental conditions or by the internal state of the robot:

Light Sensors. Heavily affected by room lighting (ambient light), unless extremely well-shielded.

Motor Force Sensing. Dependent on battery voltage. When battery weakens, force readings increase.

Light Sensors

Any light sensor will operate differently in different amounts of ambient (e.g., room) lighting. For best results when using light sensors, they should be physically shielded from room lighting as much as possible, but this is not usually perfect. Given that room lighting will affect nearly all light sensors to some degree, software should be designed to compensate for room lighting.

When using reflectance-type or break-beam light sensing, a good strategy is to be able to control the sensor's own illumination source. If a sensor reading is taken with the sensor's own illumination off, the reading due to ambient light is measured. If then a reading is taken with the illumination on, the result is a value due to ambient light plus the sensor's own illumination. By subtracting these two values, the sensor reading due to its illumination alone can be obtained.

Still, this method will not wholly eliminate the influence of ambient light. Further calibration in an actual performance environment will probably be necessary. For this, the use of global variables and an initial calibration routine is suggested.

Motor Force Sensing

Ideally, a direct measurement of the battery voltage could be used in a function to eliminate its effect on the motor force readings.

However, a simple calibration sequence might suffice. A routine could be written to take motor force readings while the robot was in motion. By manually providing resistance to the robot (and pressing a user button when ready), the resistance that the robot should interpret as "being stuck" could be directly measured.

This calibration sequence would need to be performed periodically over the life cycle of the motor battery.

6.2.3 Using Persistent Global Variables

Persistent global variables are a special type of global variable that keep their state in between pressing reset or turning the robot on and off. These are ideal for use in keeping track of calibration settings: after calibrating the robot once, it would not need to be recalibrated until after a new program were downloaded (in general, downloading of code will destroy the previous values of a persistent global, although this can be circumvented as explained in Section 7.4.3).

In order to use persistent globals, an initialization program would be created that allowed interactive setting of the values of the persistent variables. A menuing program could be written to use the two user buttons (Choose and Escape) and the "frob knob" to navigate around a series of menus. This program would allow the selection and modification or calibration of any of a number of parameters.

By exiting the initialization program without making any changes, or by simply not calling it at all, the robot can operate under the previous settings made to the persistent variables.

The routine could also allow restoration of the default values of all of the globals, returning them to some tested and known-to-work values.

6.3 Robot Control

The section will present some ideas about designing software that will control a robot. The perspective is not on low-level coding issues, but high level concepts about the special situations robots will encounter, and ways to address these peculiarities. The approach taken is to propose and examine some architectures of the control software that will be the brains of the robot.

Probably the biggest problem facing a robot is the flakiness of the overall system. A robot might face any combination of the following failure modes:

- **Mechanical Failures.** These might range from temporarily jammed movements to wedged geartrains to serious mechanical breakdown.

- **Electrical Failures.** Hopefully one can assume that the computer itself will not fail, but loose connections of motors and sensors are a common problem.

- **Sensor Unreliability.** Sensors will provide noisy data (data this is sometimes accurate but sometimes not) or simply data that is incorrect (a touch sensor might fail to be triggered).

The first two of these problems can be minimized with careful design, but the third category, sensor unreliability, is worth looking at more closely. Let's delve into a brief analysis of the sensor problem and then return to discuss control ideas.

6.3.1 Sensor Unreliability

A variety of problems afflict typical robot sensors:

Spurious Sensor Data. Most sensors will occasionally generate noise in their output. For example, an infrared sensor might indicate the presence of infrared light when there actually is no light present. Or a bend sensor might give a spurious reading.

If the noise is predicable enough, it can be filtered out in software. The noisy IR sensor might not be trusted until it gives some number of consecutive readings in agreement with one another.

However, if the noise problem is very bad, a sensor might be rendered useless, or worse, dangerous, if the program running the robot imbues too much trust in the sensor reading.

Missed Sensor Data. Affiliated with the problem of noisy data is missed data. Suppose, for either electrical or software reasons, a sensor reading is not detected. A light sensor changes state twice before the software can count it. Or a touch sensor jams and fails to trigger.

Corrupted Sensor Data. As discussed in the previous section on calibration, sensor data can be adversely affected by ambient environmental conditions or battery strength.

To some extent, unruly sensor data can be filtered or otherwise processed "at the source," that is, before higher-level control routines see it. We'll use the `wall_distance()` introduced at the beginning of the chapter as an example.

The wall distance routine got its data directly from the bend sensor and then outputted an interpretation of that data. The routine did not process the sensor data in any way—it did not check it for unreasonable data samples, for example. Suppose that the bend sensor should never report a value above 250, and for some reason, a bogus value is detected. This probably indicates some type of sensor failure—maybe the sensor is unplugged. It would make sense to intercept this failure locally, where the sensor data is first entered into the software system.

In a similar way, sensor data could be averaged, smoothed, or otherwise processed before interpretation. It is logical to assign individual routines to perform this activity for variety of sensor, which might need to be dealt with in a particular way. Using the multi-tasking capabilities of IC, each sensor or sensor sub-system could be assigned its own C process to perform this activity.

6.3.2 Task-Oriented Control

With all of these problems facing a robot, how can it get anything done? Usually—and this is not a tart response—by assuming that they don't exist. The insidious part is that most of the time, ignoring the failure modes will work. However, when the failures do occur, they will return to inflict crippling damage to a robot's performance.

Let's return again to the wall-following example, as implemented by the function `follow_wall()` (see Figure 6.3). In a worst-case scenario, what could happen while a robot was merrily running along, following a wall?

1. The robot could run into an object or a corner, properly triggering a touch sensor.
2. The robot could run into an object or corner, *not* triggering a touch sensor.
3. The robot could wander off away from the wall.
4. The robot could slam into the wall, get stuck, and conditionally trigger a touch sensor.
5. The bend sensor could fall off its mount, causing a series of incorrect sensor readings.
6. Other failures of a catastrophic nature.

Ideally, control software should *expect* occurances of cases like those numbered #1 through #4 and be able to *detect* case #5. Probably there isn't much that can be done about case #6, catastrophic failure.

Suppose we treat the wall-following activity as a discrete *robot task*, with initial conditions, an activity to perform (perhaps repetitively), exit conditions, and a return value. If we organize our thinking about robot activity with these categories, we would analyze the wall-following function as follows:

Task Analysis of Simple Wall Follow Function.

Initial conditions. Robot is in close proximity to a wall near its bend sensor.

Activity. Feedback loop with three states based on wall sensor reading.

Exit conditions. None; never exits.

Return value. None, even if it did exit.

Exit Conditions

Using this framework, we can to extend the simple wall-following function so that it could deal with several of the problems it might face while following a wall. Some of these "problems" actually *must* be dealt with; if a robot doesn't run into an obstacle sooner or later, either something is wrong, unless the robot is following a very long (circular?) wall!

```
int NORMAL_EXIT= 0;     /* exit condition code */

int follow_wall2()
{
    while (!robot_stuck()) {
        int state= wall_distance();

        if (state == TOO_CLOSE) veer_away_from_wall();
        else if (state == TOO_FAR) veer_toward_wall();
        else drive_straight();
    }

    return NORMAL_EXIT;

}
```

Figure 6.6: Wall Following Function with Exit Condition

By adding a test for touch sensors inside the loop code of `follow_wall()`, we can create a function that will detect collisions and exit. Our new function, `follow_wall2()`, is shown in Figure 6.6. Note the new sensor function `robot_stuck()`, which is expected to return a boolean true if it believes that the robot is stuck. This function can use any of the sensors available to the robot—including touch sensors, shaft encoders, and motor force sensing—to determine if this is the case.

Timeouts

Detecting collisions can only be as good as one's collision sensors. Based on the premise that such sensors are not perfect, it is a good idea to add some kind of timer-based exit condition. This will prevent the all-too-frequent case of a robot that is stuck, but does not have its touch sensor depressed. Many a robot simply does not "believe" that it is stuck—its program stays stuck in some loop, taking no compensatory action.

Time-outs can solve this problems, and provide other information as well. In a typical application, the *maximum time* that the robot is allowed to take in performing a particular task would be determined. When the function to perform the task is invoked, it would be given this maximum time. If (continuing the example) the wall-following task failed to exit before the time limit had expired, the timeout would trigger, and cause the function to exit (with an appropriate error return value).

Further, the timing information could be used to *verify* that the task had exited normally. Suppose a robot should take six seconds to get from the start of the wall-follow to running into a particular wall. If it ended up taking only three seconds, then probably an obstacle caused the premature exit.

Figure 6.7 is a listing of our third wall-following function, with timeout capability. Note that the timing variables are defined in long integers (with timing units of milliseconds). Floating point variables could also have been used (with the more intuitive units of seconds), but long integers are significantly more efficient.

If we do a task analysis of the `follow_wall3()` function, we will see a much better set of specifications, as shown in Figure 6.8.

Monitoring State Transitions inside a Feedback Loop

With the third version of the wall-following function, we can be sure that the robot will not wedge and get stuck forever. However, with a little more analysis, we can create a program to detect failure situations in advance of the overall task timeout.

The key is in the guts of the feedback loop, associated with the functions `veer_away_from_wall()`, `veer_toward_wall()`, and `drive_straight()`. When the robot is following a wall normally, these functions should alternate control, each being operative for only a short period of time. Said another way: the robot will not simply drive straight for a long time: it will veer into the wall for a bit, veer away from the wall for a bit, drive straight for a bit, etc. Contrarily, if the robot wandered away from the wall, then the `veer_toward_wall()` output would be continuously asserted.

By monitoring for normal exchange of control amongst these feedback outputs, we can detect normal operation of the feedback loop. Conversely, by looking for abnormal exchange of control—in particular, one output being asserted for too long a period of time—we can detect failure conditions.

```
int NORMAL_EXIT=    0;      /* exit condition codes */
int TIMEOUT_EXIT=   1;
int PREMATURE_EXIT= 2;

/*
  arguments to function are:
    timeout value
    maximum normal exit time
*/

int follow_wall3(long timeout, long normal_exit_max)
{
    long entry_time= mseconds();
    long exit_timeout, normal_exit_timeout, exit_time;

    exit_timeout= entry_time + timeout;
    normal_exit_timeout= entry_time + normal_exit_max;

    while (!robot_stuck() && (mseconds() < exit_timeout)) {
        int state= wall_distance();

        if (state == TOO_CLOSE) veer_away_from_wall();
        else if (state == TOO_FAR) veer_toward_wall();
        else drive_straight();
    }
    exit_time= mseconds();

    if (exit_time > exit_timeout) return TIMEOUT_EXIT;

    if (exit_time < normal_exit_timeout) return PREMATURE_EXIT;

    return NORMAL_EXIT;

}
```

Figure 6.7: Wall Following Function with Exit and Timeout Conditions

Initial conditions. Robot is in close proximity to a wall near its bend sensor.

Input values. Maximum length of time to perform wall-follow, normal length of time for wall-follow.

Activity. Feedback loop with three states based on wall sensor reading.

Exit conditions. "Robot stuck" function. Timeout.

Return value. Normal exit, timeout error exit, premature error exit.

Figure 6.8: Task Analysis of Wall Follow Function with Sensing and Timeout Capability.

The code to implement this idea works as follows: each time a new feedback output is selected, a timer is reset. The timer keeps track of how much time is spent in consecutive selections of the feedback output. If the same feedback output is selected repeatedly for too long a period of time, an exit error condition is generated.

Several constants are used to adjust the parameters of the timeout: `DRIVE_STRAIGHT_MAXTIME`, `VEER_IN_MAXTIME`, and `VEER_OUT_MAXTIME`. Three new exit error conditions are used to report which part of the feedback loop failed. The final program, `follow_wall4()`, is shown in Figures 6.9 and 6.10.

There is an annoying possibility when using feedback state monitoring that should be noted. It is conceivable that for some duration of time that it would be *correct* for the loop to stay in one state for an unusually long period of time, in which case method would incorrectly cause a premature exit. In our wall-following example, perhaps if the robot were to drive very straight and were oriented exactly parallel to the wall, it would be proper to stay in the "drive straight" state for a long while.

Upping the timeout values of the constants would minimize the potential of this problem, but would also diminish the effectiveness of the method. It is probably best to deal with these circumstances on a case-by-case basis. One possibility is to deliberately handicap the feedback control so that it oscillates a bit; clearly this has disadvantages too.

6.3.3 Coordination of Tasks

The robot task model just presented should prove useful as a way to make a robot's behavior more reliable. Yet, a further question arises: How should the selection and execution of different tasks be done?

This question is often asked by contemporary robot researchers. In addition to a

```
int NORMAL_EXIT=         0;     /* exit condition code */
int TIMEOUT_EXIT=        1;
int PREMATURE_EXIT=      2;
int DRIVE_STRAIGHT_EXIT= 3;
int VEER_IN_STUCK_EXIT=  4;
int VEER_OUT_STUCK_EXIT= 5;

long DRIVE_STRAIGHT_MAXTIME= 3000L;    /* values in milliseconds */
long VEER_IN_MAXTIME=        3000L;
long VEER_OUT_MAXTIME=       3000L;

/*
  arguments to function are:
    timeout value
    maximum normal exit time
*/

int follow_wall4(long timeout, long normal_exit_max)
{
    long entry_time= mseconds();
    long exit_timeout, normal_exit_timeout, exit_time;

    long state_start;           /* keeps track of when a state was
                                   first selected */
    int feedback_state= 0;      /* keeps track of which state is being
                                   timed */


    exit_timeout= entry_time + timeout;
    normal_exit_timeout= entry_time + normal_exit_max;

    while (!robot_stuck() && (mseconds() < exit_timeout)) {
        int state= wall_distance();

        if (state == TOO_CLOSE) {

            /* check for timeout */
            if (feedback_state == VEER_IN_EXIT)  /* reuse error codes
                                                    as state codes */
              if (mseconds() - state_start > VEER_IN_MAXTIME)
                return feedback_state;    /* exit task with error code */
            else {
                /* set state, state_start variables */
                feedback_state= VEER_IN_EXIT;
                state_start= mseconds();
            }

            /* generate movement command */
            veer_away_from_wall();
        }
```

Figure 6.9: Wall Following Function with Exit, Timeout, and Feedback Monitoring Conditions, Page 1

```
        else if (state == TOO_FAR) {
            if (feedback_state == VEER_OUT_EXIT)
              if (mseconds() - state_start > VEER_OUT_MAXTIME)
                return feedback_state;
            else {
                feedback_state= VEER_OUT_EXIT;
                state_start= mseconds();
            }
            veer_toward_wall();
        }
        else {
            if (feedback_state == DRIVE_STRAIGHT_EXIT)
              if (mseconds() - state_start > DRIVE_STRAIGHT_MAXTIME)
                return feedback_state;
            else {
                feedback_state= DRIVE_STRAIGHT_EXIT;
                state_start= mseconds();
            }
            drive_straight();
        }
    }

    exit_time= mseconds();
    if (exit_time > exit_timeout) return TIMEOUT_EXIT;
    if (exit_time > normal_exit_timeout) return PREMATURE_EXIT;
    return NORMAL_EXIT;

}
```

Figure 6.10: Wall Following Function with Exit, Timeout, and Feedback Monitoring Conditions, Page 2

variety of ways of thinking about robot tasks, there are many different approaches to organizing the higher-level control of mobile robots.

6.270 robots have special requirements in comparison to robots used more generally in research. 6.270 robots must be fast; many research robots can sit and compute for a while. 6.270 robots must be reliable; for other demonstrations, a robot might be videotaped until it performs as desired. 6.270 robots have only a few chances to perform correctly for the contest; in some research experiments, software robots are "evolved" through many generations until they behave appropriately.

Still, some of the ideas from the research field may be helpful. To extend from the task model developed in this chapter, here are several different approaches that could be used to coordinate and control task execution.

Task Sequencing

In this model, only one task executes at a time. A "task manager" is responsible for selecting tasks based upon a predetermined sequence of tasks, with alternative sequences to deal with exceptional circumstances. This can be visualized as a connected graph of tasks, with the path of traversal determined by the task manager.

A simple example of task sequencing would be a program to make a robot follow the inside wall of a rectangular area. The task manager would alternately invoke "follow wall" and "negotiate corner" tasks, assuming there were no errors.

Concurrent and Non-Competing Tasks

This model builds on the task sequencing model by allowing concurrent execution of tasks that are essentially non-interfering. For example, a task to control a "radar dish" sensor (that locates sources of infrared light) can be operated independently from a task that drives the robot. There may be some communication between the tasks (the radar dish task may wish to know that the robot's base is moving), but there is no direct interference or need for coordination between the tasks.

Concurrent and Competing Tasks

In the most general situation, concurrent tasks might interfere with each other, or compete for resources on the robot (such as control of drive motors or control of an active sensor). In this case, some method, either explicit or implicit, must be devised to resolve resource conflicts.

One method is to provide each task with a priority level; if two or more tasks are competing for the same resource, the task with the highest priority would win. A method for dealing with ties would be needed as well.

Robot Metacognition

A sophisticated task manager might have a separate module that acts as its "overseer." To use Marvin Minsky's idea and terminology from his *Society of Mind*, the main part of our brains (the "A brain") might be observed by a separate part of the brain (the "B brain").

The B brain, or overseer, checks the A-brain (which is mostly in control) for things like non-productive loops and other wedged conditions. If it detects one of these undesirable states, it makes an intervention that will provoke a different response from the A brain.

Here's an example to bring this metaphor back to our robots. Suppose a task manager (of the sequencer variety) is trying to drive the robot around the inside of a rectangle, as suggested earlier. It is alternating between two tasks: a "follow wall" task and a "negotiate corner" task. But suppose the corner routine is failing, and unbeknownst to the sequencer, the robot is stuck in the same corner.

The robot's "B-brain controller" might notice an especially tight loop between the execution of the two tasks (much the same as the wall-follower would notice trouble in shifting between the feedback outputs). The B-brain would conclude that something had gone wrong, and execute an emergency "get unwedged" task.

Control of a 6.270 Robot

In the past, most 6.270 robots have been designed with the "task sequencer" model in mind. There have been some cases of concurrent, non-competing tasks, but few or no examples of concurrent and competing tasks.

Rather than advocating a specific method, it is left for the reader to think about these issues and decide what is best for his or her own robot. Some of these methods were developed to make robots more "creature-like," which is not necessarily a desirable characteristic for a 6.270 robot. It would not be ideal for a robot to suddenly decide that it did not really want to follow that wall or whatever. A task sequencing method, perhaps with a few checks for unproductive loops, will probably be more than adequate for most robot designs.