Chapter 8

Robot Control

To the uninitiated, the term "robot" conjurs up images of a self-contained intelligent machine that completes tasks and makes decisions about it's environment. Unfortunately, technology has not quite caught up to society's far seeing prophets. The 6.270 robots that you are constructing are capable of fulfilling some of the requirements mentioned above, given some clever programming and some manner of sensors with the environment. The topic of control deals with how the robot is programmed to allow it to deal intelligently with the immediate environment.

8.1 Types of Control

The most obvious way of controlling a robot is to give it tasks to do, without any concern about the environment around it. This form of control is called *open loop* and consists of a simple signal being given and an action being carried out. The robot does not make any sort of "check" as to whether the correct action was completed, rather it mechanically follows the steps in a prescribed pattern. *Closed loop* control involves giving the robot *feedback* on the task as it progresses to allow it to acertain whether the task is actually being completed. Consider the example of a car: open loop control would consist of placing a brick on the accelerator peddle and locking the steering wheel, whereas closed loop control may have a driver making corrections for disturbances like curving roads and stop signs (not to mention other vehicles).

8.2 Open Loop Control

A control flow diagram of open loop control consists of some signal ("turn left") being interpretted by the microprocessor ("6811") to drive some actuators ("motors") and create some output ("moved robot").

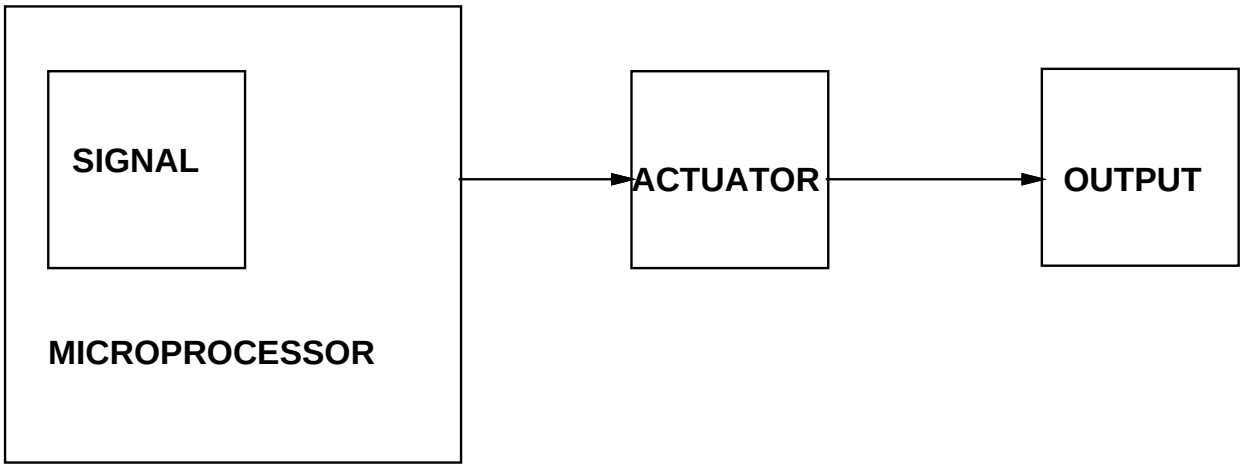

Figure 8.1: Open loop control diagram

Note that there is no link between the output and the moved robot: the microprocessor has no idea of where the robot actually is. This is often what leads to robots getting caught doing silly things at inopportune times. Pattern followers – robots that only use open loop control – often work extremely well in situations that vary little and are readily reproducible. However, the 6.270 tables (during a contest) fit neither of these specifications.

There are a number of reasons that open loop control is used rarely in the real world. The biggest one is the lack of robustness: any small change in the robot or the environment are not corrected for. Consider our haplass automobile, in it's current incarnation it is not capable of negotiating turns. Let's create a slightly smarter open loop controller – a 6.270 board hooked up to the relevant control, perhaps the accelerator peddle, brake, and steering wheel. Sending it to the store to pick up some milk would consist of programming a series of commands that negotiate the path.

backwards for 10 seconds (exit driveway)
left for 5 seconds (get parallel to the road)
straight for 30 seconds (to the stop sign)
wait for 5 seconds (a pretense of waiting for the route to be clear)
.
.
.
left for 5 seconds (pull into parking space)

There are several major problems apparent in this fictional trip, even ignoring the possibility of other vehicles on the road. If the tire pressure were low (or the wheels

were of a different size than expected) or the fuel was low (or batteries were at low power) the vehicle would start each of the tasks out of the correct position. The left hand turn may place it onto the lawn instead of the street. Even disregarding the fact that the 6.270 board is too short to see over the dashboard, our car-robot is running a blind pattern.

6.270 contest robots run into similar problems when they are controlled in an open loop manner: when they physically stray from the location the microprocessor thinks them to be in, they are lost. Changes in the friction of the gear mechanisms, or friction with the table can seriously effect the performance of the robot. Timing for open loop control will vary with battery power. The frictional and electrical variable will always be different, so the timings will never be the same for every instance. Open loop control does not take into account the robot getting stuck at a wall or with another robot.

However, there are some instances when open loop control is useful. When the robot has no way of sensing how far it is from the desired position, relying on open loop control is the only recourse. For the novice programmer, open loop algorithms are much easier to construct than their closed loop counterparts, and for some actions (namely those that change little from run to run) all that is required. It might be good to try writing a simple open loop program to determine its limitations before going on to more advanced control.

8.3 Closed Loop Control

Much research has been done on how to "properly" control vehicles and processes using the least power and getting a close as possible to the desired orientation. For this class, the bulk of the knowledge gained in 16.30 and 6.302 is dismissed, for the microprocessor allows greater flexibility for the fledgling controls programmer.

The only change to the control flow diagram is the addition of a feedback signal from a sensor shown in figure 8.2.

The sensor is used to provide information on the current orientation of the robot. This information is used to modify the control input to the robot to correct for the errors in orientation.

A concrete example is a wall following algorithm, one similar to what most of you will be implementing to navigate the table's surface. The ideal sensors to use are the bend sensors which provide (if interpretted cleverly) the distance the robot is away from the wall. Place one of these sensors on the front right side of a robot and bend it slightly so that it always bends in the same direction.

Now if the robot is too close to the wall, we would like it to move away; if it is too far it needs to move closer. Perhaps there is also some ideal range of distances from the wall in which the robot will merrily travel straight. The code to implement this

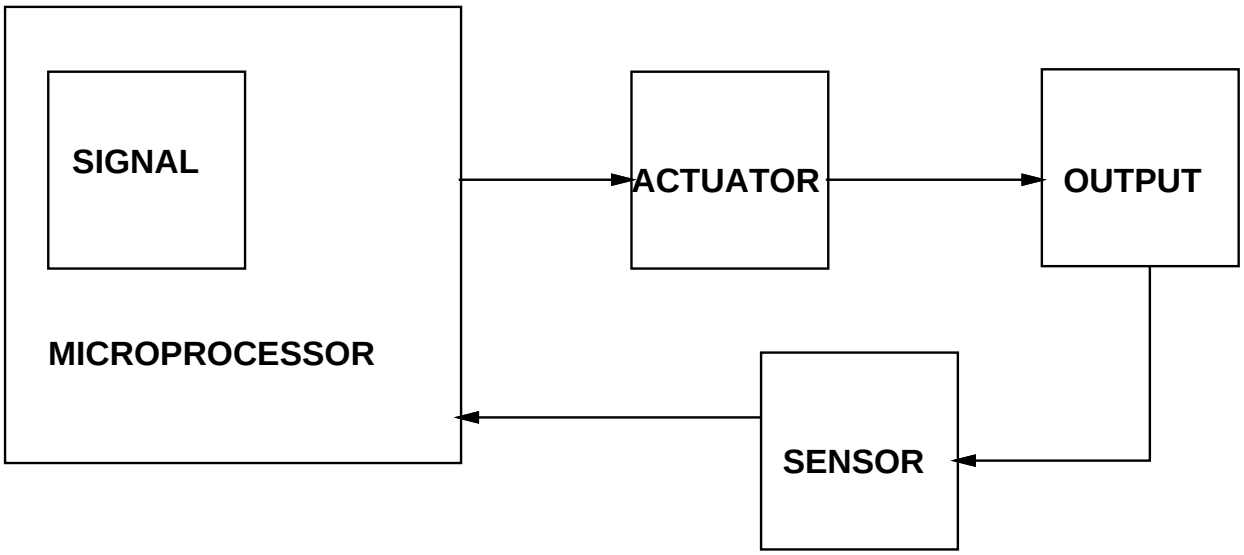

Figure 8.2: Closed loop control diagram

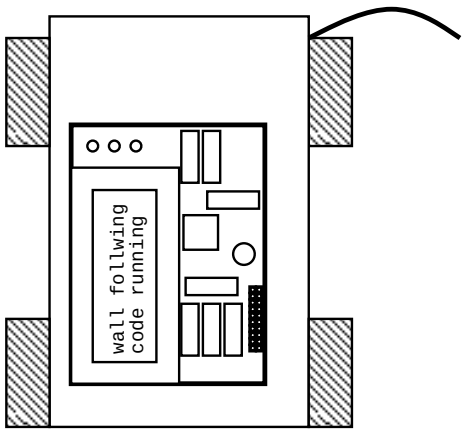

Figure 8.3: Robot with a Bend Sensor

algorithm is fairly straightforward, of more concern is how to determine the necessary values: **CLOSE** and **FAR**. In most cases, trial and error is easiest, placing the robot the correct distance from the wall, checking the sensor value, then selecting some values nearby for **CLOSE** and **FAR**. If the value of the bend sensor drops below the **FAR** threshold, the robot needs to move closer to the wall, perhaps by running the left motors slightly faster than the right motors. The opposite should occur if the sensor value climbs above **CLOSE**. If the value is in the "dead zone", the robot should continue straight.

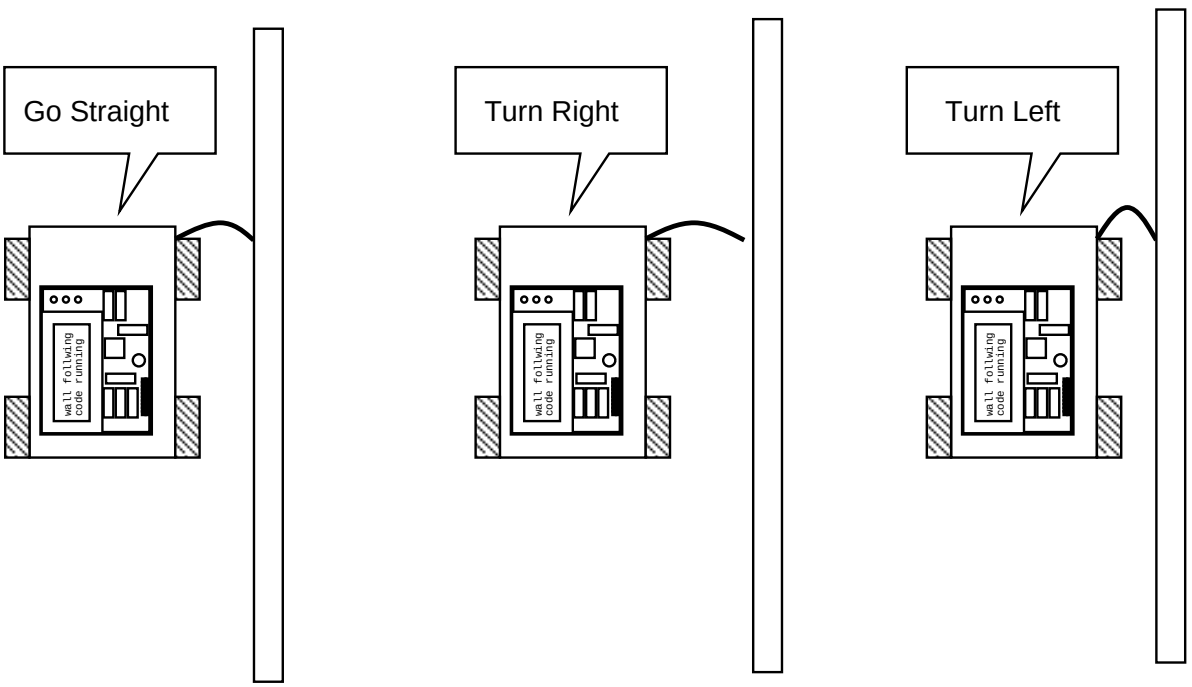

Figure 8.4: Robot moving

There is one problem that we've overlooked in creating this wall folling algorithm, which is that the robot does not simply translate towards the wall when asked to move closer. Rather, it rotates. This can lead to the robot jamming into the wall by over-rotating shown in figure 8.5.

The reason that this particular configuration can jam is that we have no feedback as to the orientation of the robot relative to the wall. It may be wiser to measure no only the distance from the wall, but also the angle relative to the wall. Placing another sensor on the back end of the robot is sufficient. Now the distance from the wall can be determined by averaging the two values from the sensors and the orientation of the robot is some function of the difference of the two sensor values. The angle of the robot with respect to the wall can be determined from which the robot can determine whether or not it is running towards or away from the wall.

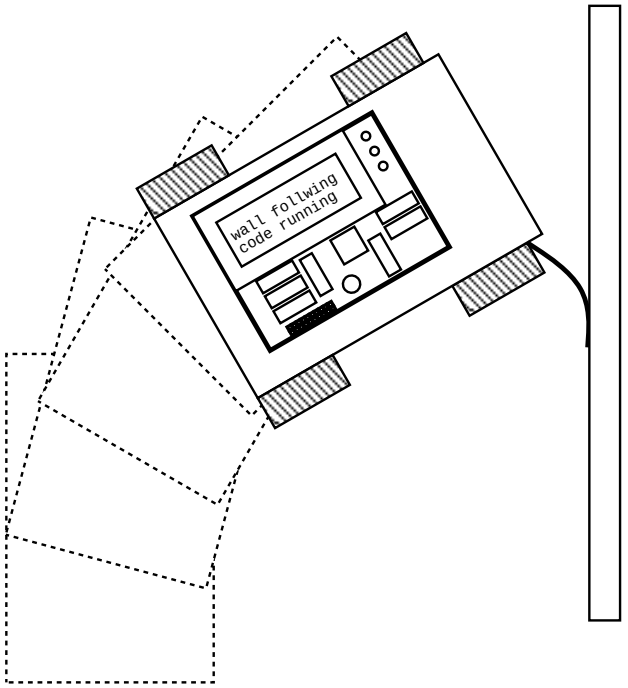

Figure 8.5: Jammed robot

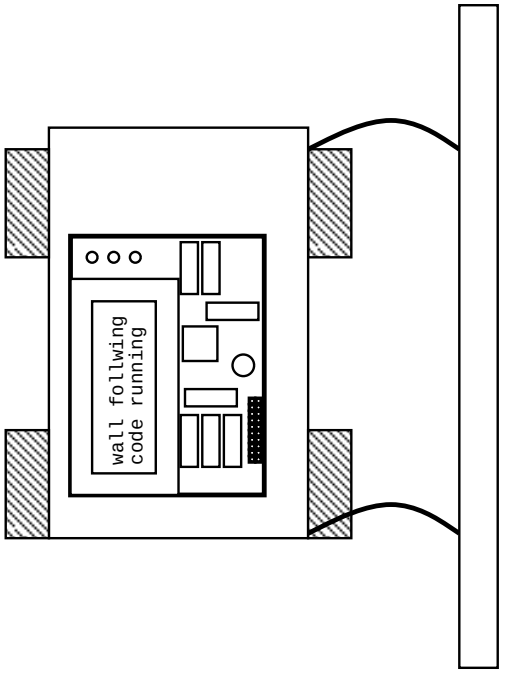

Figure 8.6: Robot with two bend sensors

Note also that the code needs to be modified to handle this extra information coming in. One easy way is to have three steps to wall following:

- Approach the wall with the front sensor until the front sensor is defected.
- Align the robot so that it is parallel to the wall. This step may be very oscilatory where the robot is moving back and forth towards and away from the wall. This is a typical problem in control if the system is not designed carefully.
- Once the machine is aligned it should proceed forward while correcting small deviations in alignment.

A very simple way of implementing this is through a group of **IF-THEN** statements. SInce each sensor can be in 3 possible states: not touching, too close, just right, there will be a total of 9 possible states. Try to make simple code using 9 if statements to drive your machine along a wall.

These are only suggestions, however, since many people come up with much more clever ways of following walls. You may wish to use previous information, like at what angle did you approach or leave the wall to add more intelligence to your code.

8.4 Closed Loop Control with Coarser Sensors

The bend sensors available in the 6.270 kit are relatively sensitive analog devices; the values correspond to the degree of bend along the length of the strip. Other available sensors are digital in nature, either because of inherent digital attributes (touch switches) or because of the manner in which they are used (breakbeam sensors).

Touch switches are very useful in situations where the optimal orientation either easily obtainable, or only required for a short time. For instance, a gate that needs to be raised to a specific height could be stopped by a touch sensor limit switch. The exact height of the gate is not of too much importance.

Reflectance sensors are inherently analog, however the thresholding done on them to determine the surface colour renders them virtually digital. Most code is not concerned with the actual shade of the surface, only whether it is white or some other colour (usually green or black). Setting the values that are considered green and black is a similar process to determining how close or far the robot should travel from the wall; the robot samples values of table surface (both green and white), places them in persistent variable and uses these for reference during the run. The difference is that these values are much more suspect to outside disturbance. Lets find out what problems may arise.

8.4.1 Analog Sensor Problems

Sensors are susceptible to a host of problems, but the somewhat controlled run situations of the 6.270 contest help to compensate for these errors. Below is a diagram of a few of the different sorts of poor data that can be read off of a sensor.

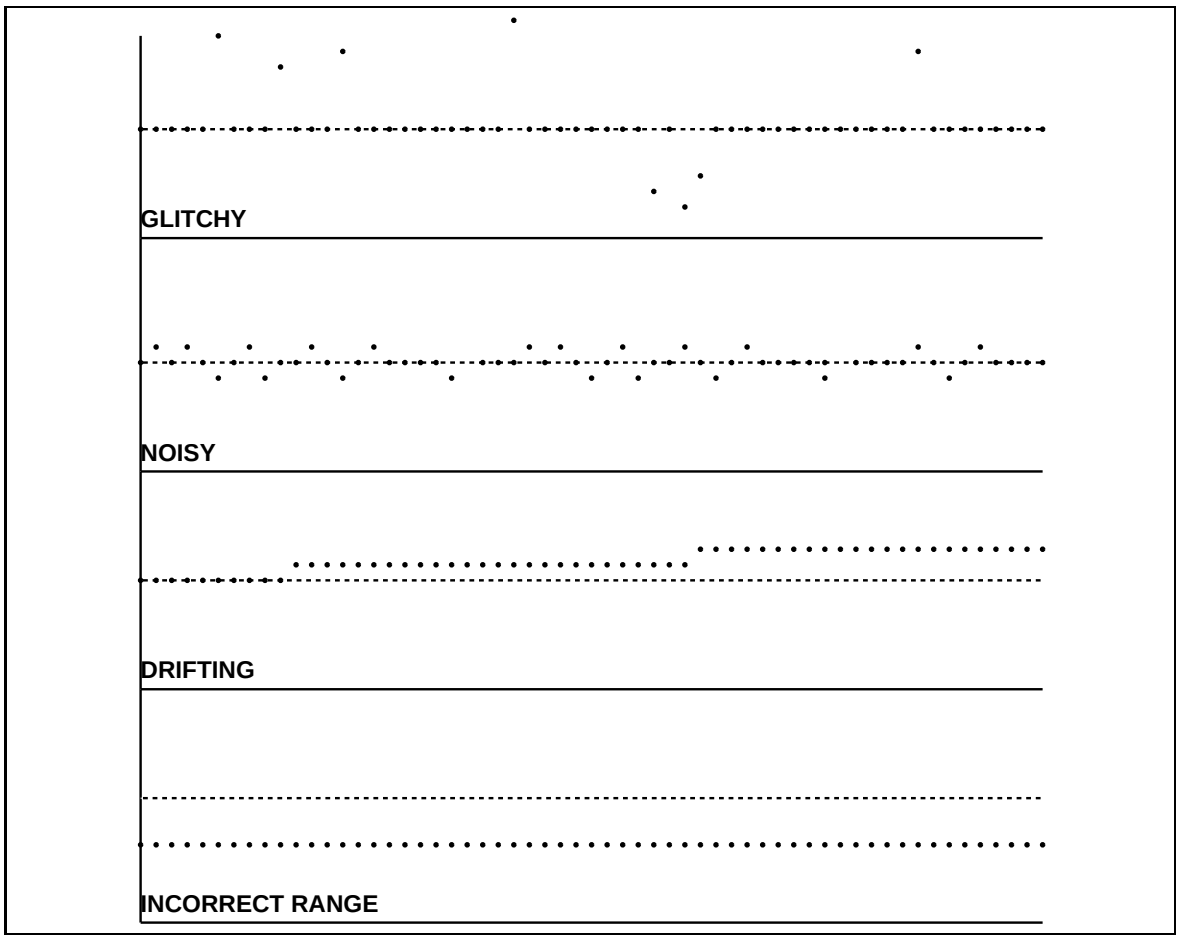

Figure 8.7: Sensor Problems that can arise.

The first problem, that of glitchy data is often inherent in the sensor itself, and must be fixed in software. Throwing out values that are out of the usable range, or looking for unlikely transitions (for instance, the robot suddenly moving too far away from the wall) is a good way to filter out unusable data.

Noisy data also caused by the sensor itself, perhaps due to oversensitivity or due to spurious input from the sensor stimuli. Time averaging the values often leads to cleaner data, and is is reasonable given how much faster the robot electronics run than the mechanical systems.

Drifting data can be caused by sensors retaining some sort of memory, if a bend

sensor is run into a wall too hard, it may become bent and retain a different unflexed valued. Slowly changing light sources may interfere with colour tracking if the sensor is not well shielded. An option is to have the robot capable of changing it's own threshold values during the course of a run. If the robot know it is not touching anything, it could use that value as the unflexed reference for the bend sensor. Or it could recalibrate colours based on what values are sensed over known surfaces. The important thing to note here is that the robot must be certain of being on a calibration surface before changing the threshold values.

Incorrect range is seldom a problem, because the ranges are easy to change in software. If the values are simply too small to show up on the analog robot input it may be necessary to change the input signal from the sensor itself. In the case of a resistive type sensors, simply placing a smaller/larger resistor in parallel or series as necessary can move and amplify the input signal from the sensor. In the case of other types of sensors, more complex amplification electronics may be required. The expansion board on the robot is suitable for mounting an op-amp or transistor to amplify small signals.

8.5 Feed Forward Control

If it is the case that we know some of the disturbances that will appear to the robot, we can correct for them before they affect performance. Consider dropping battery power: if we had some way to sense the energy left in the battery, we could compensate when it was low by having routines run for longer periods of time (since the robot will be moving slower). If the drive train of the robot was changed late in the course (an unwise thing to do), the distance travelled in the same time may change. Rather than changing all of the values, perhaps a few key ones could be changed to allow for a correction factor.

However, feed forward control is not especially useful unless there is no time to correct for the known disturbance beforehand.

8.6 Sensor Integration

There are three major concerns that must be addressed when integrating sensors to a robot: modularity, structural integrity and ease of disassembly.

Modularity refers to how easy it is to move the mounting location of a sensor. If several special Lego bricks are required to place the sensor in the correct position, it will be difficult to experiment with optimal sensor placement.

Structural integrity is necessary to prevent the sensor from falling off the robot under contest conditions. Precariously mounted bump sensors may knock themselves

loose and render themselves useless if not well mounted. Try running the robot over a few bumps and into a few walls to judge the integrity of the sensor mounts.

When a small gear deep in the heart of the robot manages to strip itself, it is helpful to have the entire robot easily fixable. This ideology extends to sensors as well; they are more susceptible to failure than most Lego pieces.

Finally, please realize that writing control programs for an autonomous robot is no small task. One way to both get an idea of the complexity of the task and some handle on how to solve it, is to play out the code written for the robot. Have one person reading the code, making decisions based on sensor values. Another person reads the current sensor values and a third person (blind-folded perhaps) actuates the robot based on the output of the code. Note that this is analogous to the control flow diagram described early. Though it sounds simplistic, it is useful to have an idea of how limited the robot really is, and how dependent it is on intelligently written control software.

In the past, most successful robots have been mechanically completed early in the course and had over a week devoted solely to creating robust, well written code to control their behaviour. Many mechanical deficiencies can be overcome in clever software, but the converse is not necessarily true.