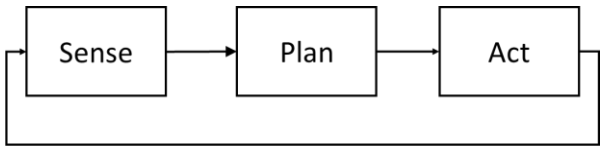
Chapter 2
Robot Control Architectures

2.1 A brief Introduction

Robots are quite complex beasts; they are a mixture of mechanical systems, sensors, computational processes and user interfaces. Moreover, robots live in a real physical (and not simulated) world, full of uncertainties and this world may change, e.g., as humans or other robots invade their space. So, you will not be surprised that the design of control systems or *architectures* can be quite involved. In this chapter we shall take a limited approach to explore architectures which you may experience on this module; we shall look at Finite State Machines, control algorithms for line following and wall-hugging (the so-called 'Bug' algorithm), and Rod Brookes 'subsumption' architecture. There are broadly speaking two approaches to robot control; the first is the **deliberative paradigm** and the second is the **reactive paradigm.**

The **deliberative** paradigm, which we won't be using, is shown in the diagram below. The robot must have an internal model of the world.



First the world is sensed, and then the internal model of the world is updated and a plan is generated. This stage is computationally heavy since it involves automated reasoning based on the internal model and sensing. Finally the robot acts. There are some problems with this model, first how to model everything the robot must know, while preventing the knowledge representation from becoming too complex. More importantly is the fact that sensing and acting are disconnected, the robot is unable to react to events such as

imminent collisions. In the **reactive** paradigm, sensing and acting are closely coupled, so the robot can react to events. There is no planning and no world model, the robot uses the various behaviours, (move forward, turn, move back) and changes between behaviours based on the sensory input. This of course assumes that we can find meaningful 'primitive' behaviours on which the robot's actions can be based. Also, we need some sort of architecture to switch between these behaviours, and that's the focus of this chapter.

2.2 Finite State Machines

Here the behaviours of a robot are associated with 'states' in a finite state machine (FSM). An example of a FSM is shown in Fig.2.1 for a hypothetical drink vending machine; here drinks cost 30p and the machine accepts 10p and 20p coins.

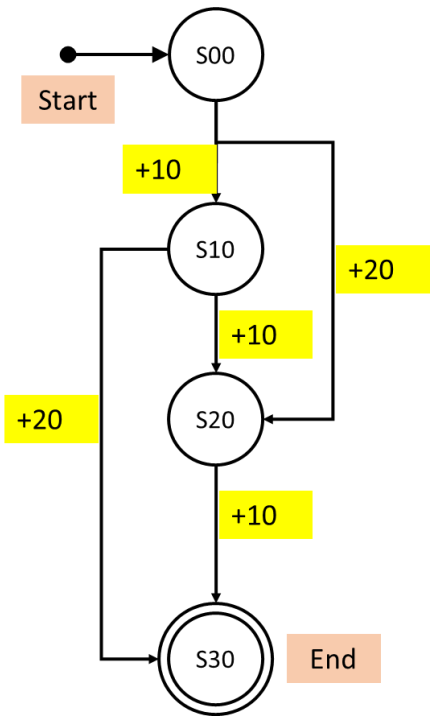


Figure 2.1 Finite State Machine for a vending machine

The state names have been chosen to tell you how much money has been entered when the machine is in that state, so S20 means that in that state 20p has been entered. You start at the top in state S00 where nothing has been entered, then entering 10p will take you to state S10, or entering 20p will put you into state S20. The yellow highlights are *transitions* between states. There are three routes to get to the end state S30, entering 3 x 10p, or one 10p then one 20p or one 20p then one 10p. You can appreciate that this *state transition diagram* fully captures the state of the vending machine at each moment.

An alternative representation of a FSM is a table, which for Fig.1 looks like this. It shows the current state, the event triggering the transition and the new state

Current State	Event	Next State
S00	+10	S10
S00	+20	S20
S10	+10	S20
S10	+20	S30
S20	+10	S30

2.3 FSM for Robot Obstacle Avoidance

Have a look at the state diagram in Fig.2.2 Each state is labelled with a useful name. The line with the blob, top left, shows the starting state. Within the body of each state are the actions taken within that state. So for STATE_FWD where the robot is told to move forward, the values of driveL and driveR are set and then these values are sent to the servos. The bottom box of each state shows the *exit condition*, i.e., what causes the FSM to transit out of STATE_FWD. We look at the distance to the obstacle, when this is below a threshold we transit to STATE_BACK.

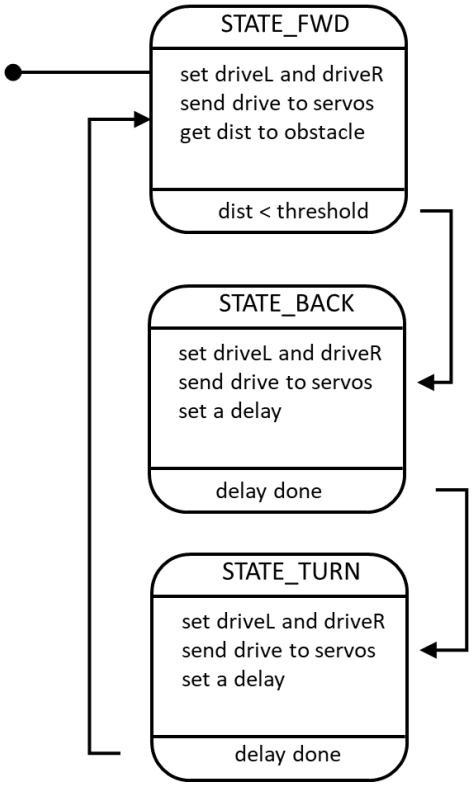


Figure 2.2 FSM for Obstacle Avoidance

In the STATE_BACK, the drives are set and sent to the servo. After a time delay which we set then we make a transition to STATE_TURN. This state also drives the servos and exits after a timeout, and then transits to the first state STATE_FWD. Coding this FSM is straightforward. First, we define the states

```
#define STATE_FORWARD 1
#define STATE_BACKWARD 2
#define STATE_TURN_LEFT 3
```

These statements are used in the pre-compilation stage where any text **STATE_FORWARD** in your code is replaced by the value 1. Then the body of the FSM, which goes inside the Arduino's **loop(){ }** block looks like this, for the first state

```
switch(state) {
    case STATE_FORWARD :
      driveL = 30;
      driveR = 30;
      driveServos(driveL,driveR);
      dist = getDistance();
      delay(60);
      if ( (dist < 300) && (dist != -1) ) {
        state = STATE_BACKWARD;
      }
      break;
```

You can see the lines to set driveL and driveR, to drive the servos, to get the distance, and the exit condition inside the

if( ) statement. Note that a dist value of -1 means that the sensor has malfunctioned. The remaining states are

```
case STATE_BACKWARD :
  driveL = -30;
  driveR = -30;
  driveServos(driveL,driveR);
  delay(3000);
  state = STATE_TURN_LEFT;
  break;

case STATE_TURN_LEFT :
  driveL = 30;
  driveR = -30;
  driveServos(driveL,driveR);
  delay(3000);
  state = STATE_FORWARD;
  break;
```

This FSM is straightforward and does its job as you will see. Let's briefly look at another FSM, this time to keep a robot travelling along a line. The robot is our usual 2-wheeled differential drive critter, with two downward facing light sensors, see Fig.2.3; each sensor reports either 1 "I can see the line" or 0, "I can't see the line. So, we have a 2-bit binary system with 4 possible states:

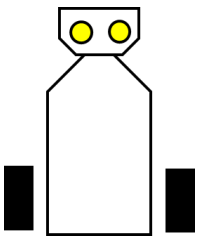
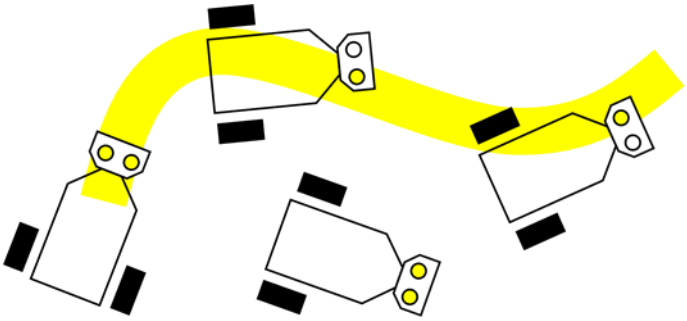


Figure 2.3 Cute 2-wheeled differential-drive robot with 2 downward facing light sensors.

Sensors		State
Left	Right	
0	0	I'm confused
0	1	I need to turn clockwise
1	0	I need to turn anti-clockwise
1	1	I'm on the line

Have a look at the diagram below, where you can easily see all four states. From the above table and diagram, it's easy to code up a FSM which will do the job.



2.4 PID Controllers

Robots are often required to follow a line, e.g., in warehouse picking operations, or in auto-drive cars, or perhaps they need to navigate around obstacles using a 'wall-hugging' algorithm. We have just discussed how to create a controller using a FSM, and while this may work in many situations, it is actually a little crude. The reason is the motors will be driven with certain values of drive, but these are *fixed* such as {driveL = 0, driveR = 40} to get the robot to rotate anti-clockwise. We need a more delicate approach, where the further the robot if off the line, the more difference in drive we send to the motors. To do this we can use a PID ('Proportional, Integral, Derivative) controller.

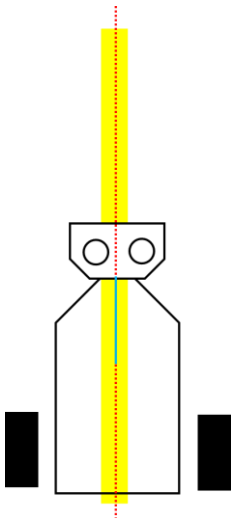
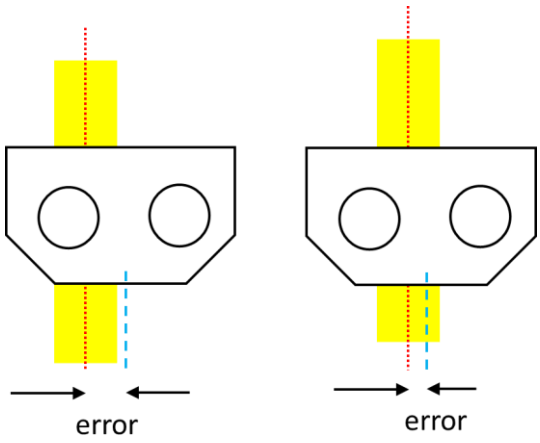


Figure 2.4 Configuration of a delicate line-follower

Before we do that, let's look at a rudimentary (but acceptable) solution the problem; the configuration is shown in Fig.2.4, the robot has to get to the centre of the yellow line (on the red dashes) and the centre of its body is shown by the blue line fixed to its body. So here it is perfect. Now let's see when things go wrong. In the diagram below on the left, the robot is too far to the right. The error is the distance between the centre of the robot and the centre of the line (red dots). The sensor system will give us this error difference which we can use to correct the robot position by setting the motor drives. In the right part of the diagram the robot is still too far to the right, but not as much and the error is less. So, it makes sense to make the motor drives *proportional* to the errors, more error, more correction, and this is correct.



So how would we go about coding this behaviour? Here's how.

```
driveR = 20 + 10*error;
driveL = 20 - 10*error;
driveServos(driveL,driveR);
```

Here the '20' part of the drive, applied to both motors keeps the robot moving in a forward direction. We add an amount 10*error to the right motor to speed it up and subtract the same from the left motor to slow it down. So, the robot will turn anti-clockwise which is what we want. But the main point is, the larger the error, the more we add and subtract, to get the robot turning. This will work, but not always.

Where has the magic number '10' come from? To understand this, think what would happen if we replaced '10' by '1'? Well, the error would have only 1/10th the effect, so the correction would not be as strong. So, the correction is *proportional* to this magic number 10. So let's replace this magic number by a coefficient **`Kp`** which stands for 'proportional coefficient'. We have just discovered the 'proportional' bit of the PID controller!

```
driveR = 20 + Kp*error;
driveL = 20 - Kp*error;
driveServos(driveL,driveR);
```

2.4.1 Why the Proportional Controller may fail

Here we shall start the development of the PID theory. Let's consider a toy problem where the robot has to move so that its lateral position is 1mm from the centre of the line. Let's try a few values of **Kp** to see how it fairs. Have a look at Fig.2.5 where the robot starts off at 0 mm and we drive it using the proportional error towards the 1 mm position (up the side of the graph). This is a graph of robot position against time, and we want the position to become 1.0, near the top of the graph. Horror! None of the values of **Kp** actually work! Also, for larger values of **Kp** the robot starts to oscillate! This is certainly not desirable. This is where PID control steps in.

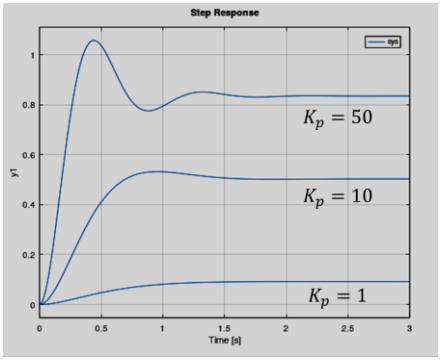


Figure 2.5 Response of robot to a disturbance with three values of Kp

2.4.2 Structure of the PID Controller.

This is shown in Fig.2.6 which shows the error signal $e(t)$ coming in at the left and the computed drive exiting at the right to drive out motors. The three boxes in the middle make different computations on the error signal, and these are summed (the Σ symbol) which form the final drive. We know how to calculate the proportional component. The integral component sums all the errors over time (this will include negative error values, so the sum will not simply increase). The derivative component takes the *difference* between current and previous errors. It turns out, applying some theory, that such an arrangement can provide good control of most systems, such as robots, self-driving cars, constant temperature heating systems, hard disk drive head positioning; you get the idea.

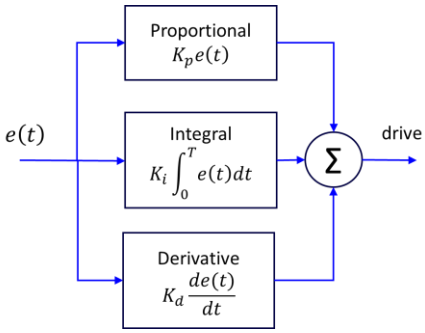


Figure 2.6 The structure of the PID controller; from top to bottom, P, I then D

So, let's see what the new components do, first the derivative. Let's stick with Kp = 50 (see Fig.2.5 where we had the horrible overshoot and oscillations) and look at two values of Kd; Kd = 0 and Kd = 10. You can see the results in Fig.2.7. With Kd = 0, we have oscillations and overshoot, but with Kd = 10, the oscillations are damped out, so the position rises smoothly towards the desired position 1.0, but still doesn't get there. That's the job of Ki as we shall now see.

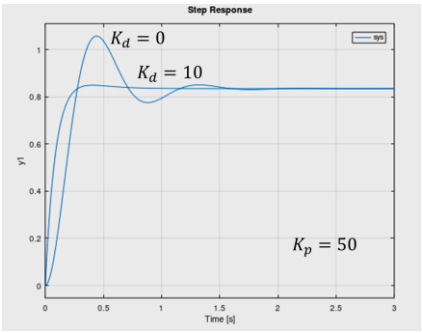


Figure 2.7 PID controller with two values of Kd

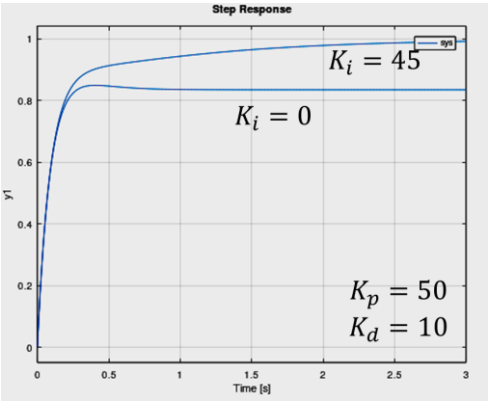


Figure 2.8 Effect of the coefficient Ki

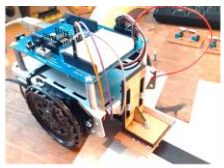
Have a glance at Fig.2.8 where the previous curve for Kp=50, Kd=10 (and assumed Kp=0) is drawn again, and the robot does not move to 1.0 mm but only manages a little over 0.8 mm. However, with Ki = 45 the results are much better, the robot is clearly moving towards the goal of 1.0mm.

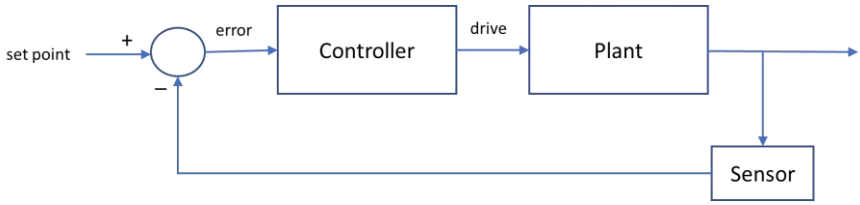
So, in summary here is what the three coefficients do.

K_p	proportional	Makes robot move to goal. Too large a value gives overshoot and oscillation. Goal is not achieved.
K_d	derivative	Overshoot and oscillation removed. Goal still not achieved
K_i	integral	Enables goal to be achieved.

2.4.3 Location of the Controller in the 'Loop'

Let's put all of this together and see where the robot fits in. Have a look at the diagram below which shows a general control loop which, as we have hinted above, can be applied to various devices, the 'plant', which is here our robot. At the left is the set point, in our example the desired robot position. At the right, a sensor monitors the actual robot position and sends this back to the start of the loop. Here the difference between desired and actual position is calculated to give the error signal which is then input into the controller.





The PID controller does its job as described above and inputs a drive signal to the robot's motors making it to move towards the desired location.

While the *structure* of the PID controller will be the same for all robots, the coefficients will depend on the particular robot in question. Let's finally think about this.

2.4.4 'Tuning' the controller

This refers to finding the values of K_p, K_d, and K_i. This is usually done experimentally and is something of a black art which is learned by experience. The usual approach is as follows.

Stage 1.	Set all coefficients to 0
Stage 2.	Increase K_p until the robot shows signs of overshoot or oscillation
Stage 3.	Increase K_d until the oscillation disappears
Stage 4.	Increase K_i until the robot achieves the desired position. This may mean reducing K_p and perhaps K_i.