Chapter 1
Robot Kinematics

1.1 A brief Introduction

Kinematics is the study of movement (well almost) so in this chapter we are looking at the principles of robot movement, for our 2-wheeled differential drive robot. The explanation is mathematical; this is necessary for us to write code to get the robot moving. The most important expressions are highlighted. So how does a wheeled robot move? Simply by driving its wheels. If you drive both wheels at the same angular speed, then the robot moves forward. If you drive the left wheel faster, then the robot will arc to the right. Driving both wheels at the same speed, but in opposite directions, will make the robot spin about its axis.

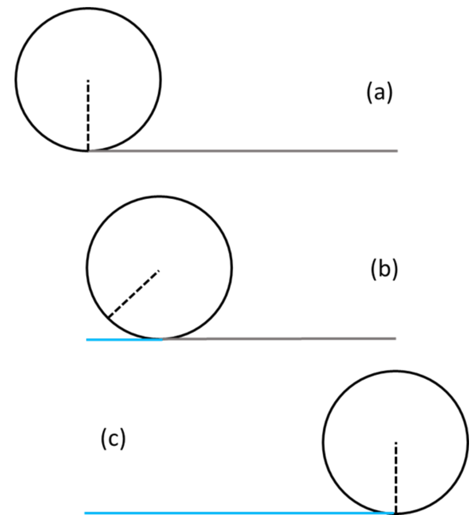


Figure 1.1. Rotating wheel leaving a track of blue paint on the surface

1.2 Linear and Angular Velocities

Let's see how wheels work. Fig. 1.1 shows a wheel completing a full revolution. Imagine the type is coated with paint, so as the wheel rotates it paints a nice line on the surface. How long is this line? Simply the circumference of the wheel.

Now let's do a quick calculation. I guess you will remember the expression for the circumference of a circle radius r. This is of course $2\pi r$. So, if we have a wheel of radius 33mm then one rotation will shift the robot a distance (2)(3.1415)(33) = 207mm.

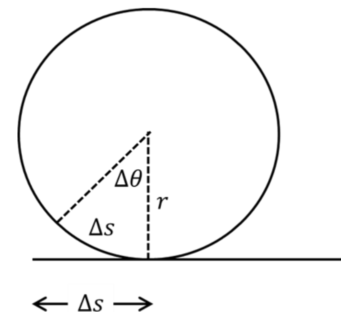


Figure 1.2. Wheel rotating with a change in angle

And now for some maths. If one rotation gives a distance of $2\pi r$ then half a rotation will give half of this distance, but what about if the angle of rotation is $\Delta\theta$? (Here the symbol Δ means a change and θ is the angle. Well, the arc of the circle corresponding to the angle $\Delta\theta$ is just

$$\Delta s = r\Delta\theta \qquad (1)$$

where the angle is in radians (more on that later). So, this is the length of the line painted by the robot, and is the distance moved forward, look at Fig.1.2.

Now let's think about the robot's speed in mm/sec. Speed is the distance moved Δs in a time interval Δt. Here's the expression for speed.

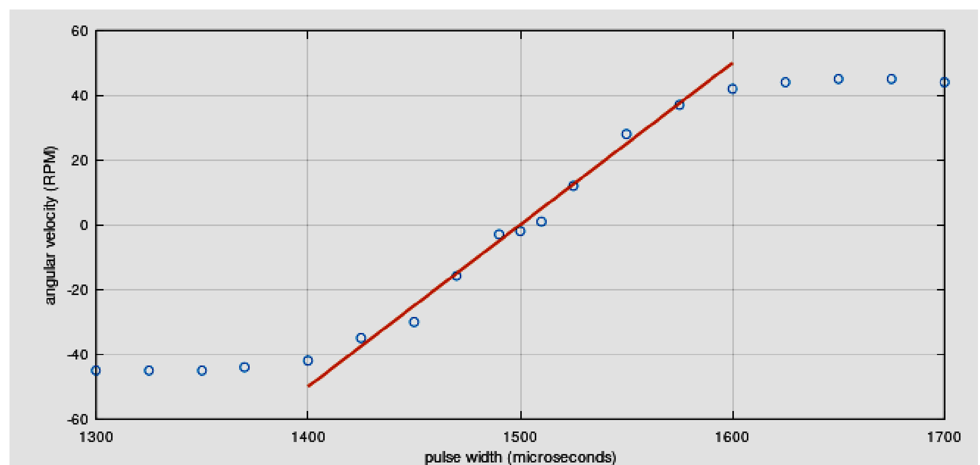
$$v = \frac{\Delta s}{\Delta t} \qquad (2)$$

Continuing with our example, if the robot wheels make one revolution in 2 seconds, then the robot speed is

$$v = \frac{2\pi r}{2} = \frac{207}{2} = 103.5\ mm/sec$$

Try to imagine what that means. The symbol for speed is v since what we are really talking about is *velocity*, speed in a particular direction (forward or backward).

But robots move by turning their wheel with their servomotors and our computer programs must provide *drive* to the motors to make them rotate. To make a servomotor rotate, we must give it a series of 'pulses' where the pulse width determines the rotational speed. For the robot we shall be using, the 'Parallax Activity Bot /BoE Bot' the relationship between pulse width and speed is shown in the diagram below. The points show real measurements, and the



red line has been placed to capture the most useful part of the experimental curve, where speed is proportional to pulse width. You can see that with a pulse width of 1500 μs (microseconds) the motor does not rotate. Therefore, in our code (and in our thinking) we shall define a quantity **drive** *relative to this 1500*. So, a drive of 50 will create a rotational velocity of around 30 rpm, and a drive of -50 will rotate the motor at this rpm, but in the opposite direction.

It's easy to get a relationship between drive and rpm from the red line above; here we find this is

$$drive = \frac{50.0}{30.0} rpm \qquad (3)$$

In our code, you will see the variables **driveL** and **driveR** which are, of course, the drives sent to the left and right servomotors.

Now we need to understand how the values of driveL and driveR will determine the robot's speed; let's assume these are the same, so the robot will move forwards (we'll look at other possibilities later). If we stick expression (1) for the distance gone when the wheel rotates through an angle, into expression (2) which is the definition for linear robot velocity moving forward, we get

$$v = \frac{\Delta s}{\Delta t} = \frac{r\Delta\theta}{\Delta t} = r\,\frac{\Delta\theta}{\Delta t}$$

Here $\Delta\theta/\Delta t$ tells us how fast the wheel angle changes with time, so this is the *angular velocity* of the wheel; we give this the symbol ω so we have a fundamental expression

$$v = \omega r \qquad (4)$$

This makes sense; if the wheel radius r is increased, then the linear speed v is increased, and if the angular velocity ω is increased (the wheel rotates faster) then the linear speed is increased. All seems good, and it is. There may be a conceptual stumbling block, however, it's due to the *units* of angular velocity ω. The angle change in (1) is measured in

radians (in one revolution, there are 360 degrees which is 2π radians, a little over 6). So, we need to connect angular velocity in radians/sec to angular velocity in rpm, so we can use the above drive graph.

Help! What are radians?

Think of a wheel rotating once, through 360 degrees. We know the distance gone is the circumference of the wheel $2\pi r$. Now expression (1) tells us that this distance is $r\Delta\theta$. So, we have

$$2\pi r = r\Delta\theta$$

and cancelling the *r* gives us

$$\Delta\theta = 2\pi$$

therefore, in a circle, there are 2π radians.

Let's say the wheels are rotating at n rpm, i.e., n_{rpm}. Therefore, the revs per second is

$$n_{rps} = \frac{n_{rpm}}{60}$$

Each rotation the wheel rotates 2π radians, therefore the radians per second (ω) is just

$$\omega = 2\pi \frac{n_{rpm}}{60} = \frac{2\pi}{60} n_{rpm}$$

i.e.,

$$\omega = \frac{2\pi}{60} n_{rpm} \qquad (5)$$

Let's work through an example how we would use this maths to make a robot move forwards a desired distance, in a desired time.

Let’s take the case of driving the robot a desired distance of 80mm in a desired time of 2 seconds, a speed of 40 mm/s.

$v = \frac{\Delta s}{\Delta t}$ (2)	*v = 80/2 = 40 mm/sec*
$\omega = \frac{v}{r}$ from (4)	ω=40/33.0 = 1.21 rad/sec
$n_{rpm} = \frac{60}{2\pi}\omega$ from (5)	n_{rpm}= (60*1.21)/6.28 = 12 rpm
	drive = (50.0/30.0)*12 = 20

We shall revisit this below when we discuss how to code our robot.

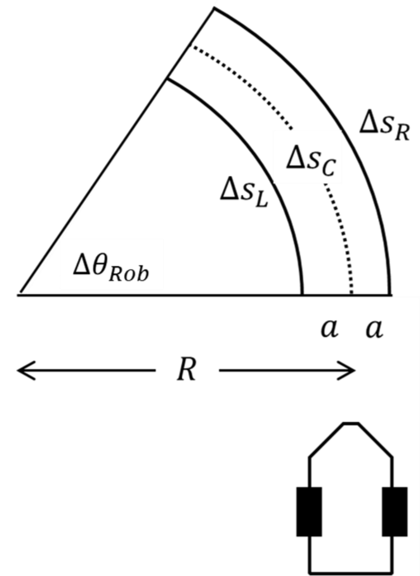


Figure 1.2. Robot moving on an arc. Distance a is between each wheel and the robot centre.

1.3 Movement on an arc

Have a look at Fig.1.3, at the bottom you will see the robot. The length of its axle, connecting its wheels is $2a$ and for the Parallax robot we are using, this is 104.0 mm, so we have a = 52 mm.

The diagram shows that the centre of the robot (between the wheels) moves along an arc of radius R. So the left wheel moves along an arc of radius $(R - a)$ and the right wheel moves along a larger arc of radius $(R + a)$. Clearly the right wheel is moving faster than the left. The robot’s *pose* changes, it started off moving North, and it ends up moving Northwest, having changed its bearing by an angle θ_{Rob}. The subscript *Rob* reminds us we are thinking about the entire robot, rather than its wheels.

To understand the maths which follows, we apply the relationship
$\Delta s = r\Delta\theta$ used above. So, for the left wheel we have

$$\Delta s_L = (R - a)\Delta\theta_{Rob} \qquad (6a)$$

and for the right wheel

$$\Delta s_R = (R + a)\Delta\theta_{Rob} \qquad (6b)$$

Now let's imagine that the robot takes a certain time interval Δt to complete its trajectory along the arc. To find the robot wheel speeds, we must divide distance gone by each wheel by this time interval, see expression (2). So, we get for the left wheel

$$v_L = \frac{\Delta s_L}{\Delta t} = (R - a)\frac{\Delta\theta_{Rob}}{\Delta t} \qquad (7a)$$

and for the right wheel

$$v_R = \frac{\Delta s_R}{\Delta t} = (R + a)\frac{\Delta\theta_{Rob}}{\Delta t} \qquad (7b)$$

This is fine, but these expressions are not really telling us much. So, we must go forwards a bit. Remember the expression (1) connecting wheel distance and angle. For the left and right wheels these become, since both wheels have the same radius.

$$\Delta s_L = r\Delta\theta_L, \quad and \quad \Delta s_R = r\Delta\theta_R$$

and putting these into expressions (7a) and (7b) we find

$$r\left(\frac{\Delta\theta_L}{\Delta t}\right) = (R - a)\left(\frac{\Delta\theta_{Rob}}{\Delta t}\right) \qquad (8a)$$

and

$$r\left(\frac{\Delta\theta_R}{\Delta t}\right) = (R + a)\left(\frac{\Delta\theta_{Rob}}{\Delta t}\right) \qquad (8b)$$

I've stuck in some brackets here, where changes in angles are divided by a corresponding change in time. These are angular velocities, speeds of rotation.

The symbol for angular velocity is **omega** ω (lower case) or Ω (upper case). Lower case ω is the angular velocity of the *wheels* and upper case Ω is the angular velocity of the robot body, when viewed from above; this is just the rotation speed of the robot.

So, we can rewrite equations (8) like this

$$r\omega_L = (R - a)\Omega_{Rob}$$

$$r\omega_R = (R + a)\Omega_{Rob}$$

therefore

$$\omega_L = \frac{(R - a)\Omega_{Rob}}{r} \qquad (9a)$$

$$\omega_R = \frac{(R + a)\Omega_{Rob}}{r} \qquad (9b)$$

These are very useful expressions. As with all expressions, think of the stuff on the right of the = sign as an *input* to a computation, what we want to calculate, and the stuff on the left is what we have to code to make this happen. So, if we want the robot to go around an arc of radius R with an angular velocity Ω_{Rob} then we have to make the motors rotate with angular velocities ω_L and ω_R.

1.4 Special Cases

There are two special cases of the maths in expressions (9). First, when the robot is moving straight, then we can write $R \to \infty$ so we can neglect a in the expressions, and R divides out. You can see this by calculating the ratio of the wheel omega's

$$\frac{\omega_L}{\omega_R} = \frac{(R - a)}{(R + a)} \qquad (10)$$

where the ratio becomes $R/R = 1$ which tells us that the omegas are the same, i.e., both wheels rotate at the same speed.

The other special case is when $R = 0$, this is where the robot rotates about its centre. Putting this into expressions (10) we find

$$\frac{\omega_L}{\omega_R} = -1 \qquad (11)$$

so, the omegas are equal and opposite!

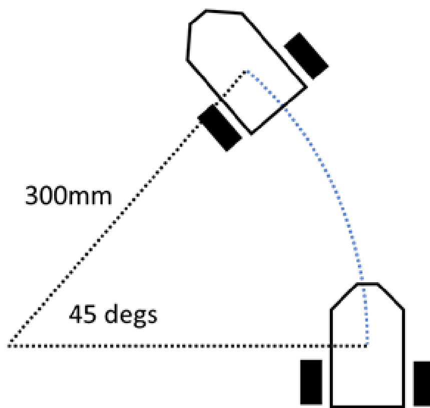


Figure 1.3 Worked example. The robot rotates 45 degs on an arc 300mm and takes 2 seconds.

A Worked Example

Let's say we want our robot to travel along an arc of radius 300mm and change its pose by 45 degrees, and it does this in 2 seconds, see Fig. 1.4.

The angle expressed in radians is $45\pi/180$ = 0.785rad. The angular velocity of the robot Ω_{Rob} is 0.785/2.0 = 0.393 rad/sec.

Since a = 52mm for the Parallax robot and our arc has a radius of 300 mm, plugging these into expressions (8) gives us

$$\omega_L = \frac{(300-52)0.393}{33}, \quad \omega_R = \frac{(300+52)0.393}{33}$$

and so, we calculate

$$\omega_L = 2.85\ rad/s\,, \qquad \omega_R = 4.19\ rad/s$$

Now we need to convert these *omegas* to *rpms*. Since an omega of 2π rad/s corresponds to 1 rev/sec, then 1 rad/s corresponds to $1/2\pi$ rev/sec. Then ω rad/s corresponds to $\omega/2\pi$ rev/sec, and therefore to $60\omega/2\pi$ rpm. To get revs per second we divide the omegas by 2π which gives us

left 0.45 revs/sec	right 0.67 revs/sec

and to get the revs/min we multiply by 60

left 27 revs/min	right revs/min	40.2

and using the expression (3) for drive, we finally have

left drive 45	right drive 67

which are the drive signals we send to our motors.

All of these calculations are done in our Arduino code. The purpose of this worked example is simply to explain what the code actually does.

1.5 How to Code a Real Robot

We need to get the robot to move forward a certain distance we specify or rotate an angle we specify or rotate on an arc. In our code, we have to specify **drives** which will make the robot move as we want, then send these drives to the servos. Here's a code snippet which gets the robot moving forwards

```
driveL = 30;
driveR = 30;
driveServos(driveL,driveR);
```

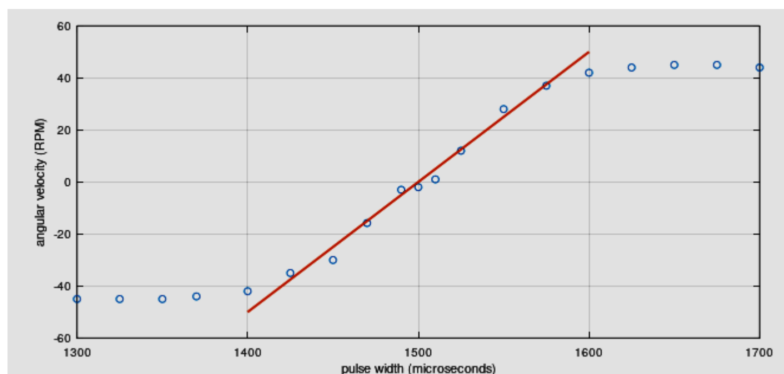
This code will get the robot moving (for ever) but we would rather like to tell the robot *how far to move* and *with what speed*. So, here's some code to get the robot moving forward for a specified distance (in mm) over a specified time. We relate the code to the corresponding maths. All the variables have been declared for you in the code templates.

`desDist = 80;` `desTime = 2.0;`	
`desSpeed = desDist/desTime;`	expression (2)
`omega = desSpeed/wheelRad;`	from expression (4)
`rpm = (60/(2*PI))*omega;`	from expression (5)
`driveL = (50.0/30.0)*rpm;`	expression (3)
`driveR = driveL;`	
`driveServos(driveL,driveR);`	
`delayTime = (int)(desTime*1000)`	
`delay(delayTime);`	
`driveServos(0.0,0.0);`	

The last three lines deserve some comment. The lines before them set the motor drives and therefore their speeds. These

last three lines specify how long the servos must spin. So we calculate the **delayTime** in milliseconds from our **desTime** (in seconds), and pass it to the **delay(…)** function which suspends the MPU for this time. Then, we drive the servos with driveL = 0.0 and driveR = 0.0 which will make them stop. So, the final result is that the robot will move 80mm forwards and it will take 2secs to do this, both values we specify.

Now the maths is perfect, and the code is perfect (they are both perfect *abstract* systems of thinking), but when you get the robot to execute this code, it will not move 80mm forwards, it will be more, or it will be less. Why? Because the robot is not *abstract*, it is not a *simulation*, it is really *embodied in the real world*. Think of its motors, their datasheet may specify their 'accuracy' as 10%. This means that if you ask them to rotate with an angular velocity of 10 rpm, they will rotate with anywhere between 9 and 11 rpm. In the worst-case scenario, the left motor could rotate at 9 rpm and the right at 11 rpm; the robot would not go straight forward but arc to the left! Also, we need to know the wheel radius, and so we measure it, but our measurements are subject to errors. And, also the relationship between **drive** and rpm may not be the one we presented earlier, each motor is different. Looking at the drive-rpm curve again (reproduced below) we see a real issue.



If we look at pulse Width 1500 (drive = 0) we see that increasing the drive to about 20 does not make the motor turn! There is a 'dead band'. This means that we cannot use small forward drives or motor speeds.

So, back to our problem. There are two ways to cope with this problem: The first relies on direct observation of the robot, and measurement of how it moves. Let's say we ask the robot to move 80mm and we measure how far it moves, 90mm; it's gone too far. So we could reduce the wheel velocity **omega**, but we know this is dangerous, since if omega is made too small, then the motor will not turn.

There is another way to make the robot travel less far; we can reduce the amount of time we drive the motors. So, we change the **desTime**

```
correctionFactor = desDist/actualDist;
desTime = desTime*correctionFactor;
```

It is important that we make this change at the appropriate place in the code, it must not be made before we calculate the omega's since they depend on the desTime. Here's where to put the correction, so that it only affects the **delayTime** which tells the motors how long to rotate.

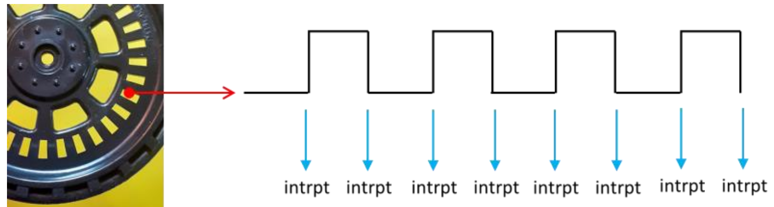
```
correctionFactor = desDist/actualDist;

desTime = desTime*correctionFactor;
delayTime = (int)(desTime*1000)
delay(delayTime);
driveServos(0.0,0.0);
```

Figure 1.4 Wheel encoder: Top the IRed transmitter and receiver measures reflections from the spokes (bottom)

1.6 Wheel encoder technology

Here we shall look at improving our work on movement using dead reckoning by employing a robot *proprioceptive* sensor which measures the position of the wheels. The sensing device is called a 'wheel encoder', see Fig.1.5. As the wheel rotates, IRed light from the transmitter shown at the top

passes through slots in the wheel, shown at the bottom. When the light hits a spoke, it is reflected to the IRed sensor and produces a pulse sent to the Arduino. So, as the wheel rotates, it sends a series of pulses to the Arduino shown in the diagram below.



The pulses arrive at Arduino pins especially configured to receive hardware 'interrupts' and when a pulse arrives, the code breaks out of its current execution place and jumps to an *Interrupt Service Routine (ISR)* and execution continues there. When the ISR is complete, then execution returns to the place where it was forced to break out from. Each wheel has an interrupt, the ISR for the right wheel is shown below.

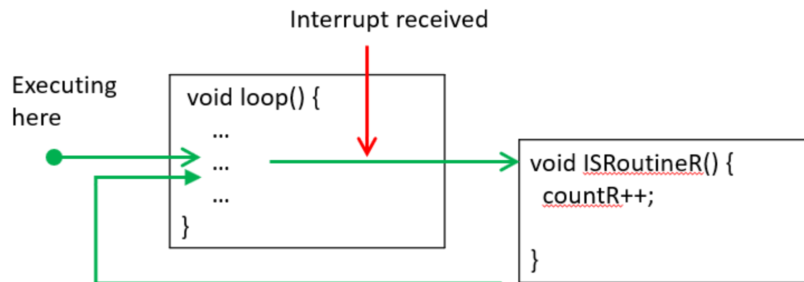
```
void ISRoutineR() {
  countR++;
}
```

You can see this ISR increments the value of **countR** i.e., every time the encoder sends a pulse, this value is incremented, so the code knows how many steps the wheel has rotated. Here's an overview of how interrupts work. First the hardware pin is attached to the ISR, and the interrupt is configured to respond to a CHANGE (i.e., a *rising* or a *falling* pulse edge. This code is in **setup();**

```
attachInterrupt(digitalPinToInterrupt(EncoderR
_pin),ISRoutineR,CHANGE);
```

Here's what happens when the code is executing in loop() and a pulse arrives. Code execution is shown by the green arrow with a blob. When the interrupt is received, execution transfers execution returns to where it left off. It is important

to understand that this occurs at the level of hardware instructions, not lines of code. It is extremely fast.



Now we need to understand how many pulses are received in one wheel revolution, and therefore how far the wheel moves between pulses. There are 32 spokes in the wheel, and since the ISR responds to both rising and falling edges, then there are 64 pulses per wheel revolution. The angle rotated between pulses is a little under 6 degrees, see Fig.1.6. But how far has the wheel moved? We know that

$$\Delta s = r\Delta\theta$$

so for a wheel radius of 33 mm, we have, calculating using radians

$$\Delta s = 33\frac{2\pi}{64}$$

which works out to be 3.24 mm. This is the accuracy of any measurement of wheel travel we can make since it is the smallest step in distance we can possibly know.

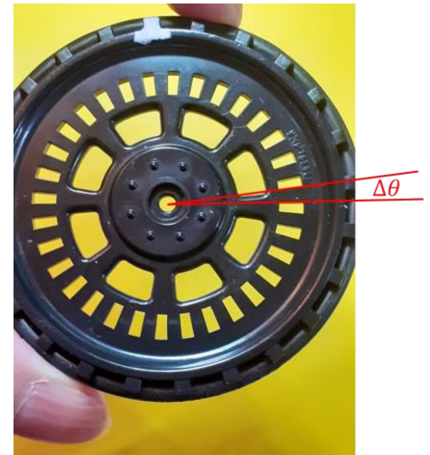


Figure 1.5 Angle shown between pair of rising and falling pulse edges

1.7 Moving on a straight line of given distance.

Let's say we want the robot to move a desired distance. How do we use wheel encoders to make this happen? First we need to calculate the number of steps required, which is the number

of pulses the encoders receive, setting **countL** and **countR**. The number of pulses is simply the distance gone divided by the step size,

$$n_L = n_R = \frac{desired\ dist}{\Delta x}$$

For example, to get the robot to move 300 mm, we need to count 300/3.24 = 93 pulses, approx. Here's some example code which will do the job. We set the motor drives and switch on the servos. Then we monitor the actual counts and when they exceed the counts we need ($n_L = n_R$) then we stop the servos.

```
driveL = drive;
 driveR = drive;
 driveServos(driveL,driveR);

 if((countL > nL) && (countR > nR))  {
   driveServos(0.0,0.0);
   servosDetach();
 }
```

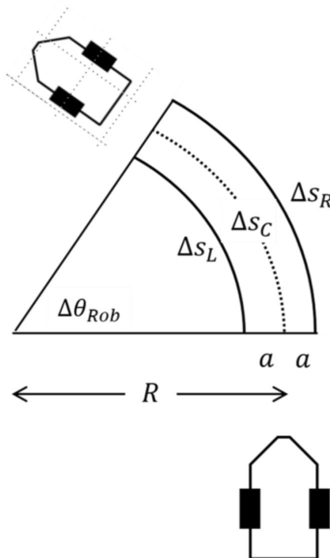


Figure 1.7 Geometry for moving on a curve with specified angle and radius.

1.8 Moving on an Arc

We've already seen the maths for this, but let's revisit it here. The arrangement is shown in Fig.1.7 where the symbol θ_{Rob} refers to the rotation of the entire robot about its centre (in the middle of its axle). The arc is specified by this angle and also the radius of the curve. Remember the robot axle has length $2a$ as shown in the diagram.

The robot's wheels travel the following distances when traversing the arc,

$$\Delta s_L = (R - a)\Delta\theta_{Rob}$$

$$\Delta s_R = (R + a)\Delta\theta_{Rob}$$

The number of pulses is calculated by dividing the distances travelled by the step size Δx. So, we get

$$n_L = \frac{\Delta s_L}{\Delta x} = \frac{(R - a)\Delta\theta_{Rob}}{\Delta x}$$

and a similar expression for n_L. All quantities on the right are known, so it is easy to calculate the number of pulses for each wheel.

There is one additional factor we need to take into account. Not only is the right wheel making more steps, but it is also moving faster. Since both wheels must start and stop at the same time, the speeds are proportional to the number of steps. So, we have where *fwd* represents some *drive* to the servos,

$$\frac{fwd_R}{fwd_L} = \frac{n_R}{n_L}$$

The following code does all of this for us, where we are asking the robot to make a 90-degree arc of radius 300 mm.

```
desRadius = 300;
desDegrees = 90;
desTheta = desDegrees*(PI/180.0);

nLf =(desRadius - axleLen/2.0)*desTheta/dx;
nRf =(desRadius + axleLen/2.0)*desTheta/dx;

nL = (unsigned long) nLf;
nR = (unsigned long) nRf;
fwdL = 20;
fwdR = fwdL*nRf/nLf;
```

1.9 Robot Actuators

Robots move using their actuators (motors). We can classify actuators in several ways. First, we can consider how they move through space; most actuators we have experienced use rotation, which attached to wheels produce linear motion through space (along a straight or along a curve). But there are also *pure* linear actuators, which move in a straight line. Rotational actuators may be placed in sub-classes; there are *dc-motors* which when connected to a 5Volt source rotate with a certain angular speed, there are *servomotors* where you control their angular speed by giving them pulses of varying widths, and finally there are *stepper-motors* which move through a discrete angle when you give them a pulse. Stepper motors are capable of very accurate (can move small distances) and repeatable motion; they find application in laser-cutters, 3D printers, photocopiers and in robotic surgery.

Recent technology provides us with extended actuator possibilities, one example is shape-memory alloy. Think of a wire which can be long or short depending on its temperature; send a current through it (to heat it up) and it could be short, stop the current (it cools) and it will be long. So, you can control its length electronically. Applications are found in limb prosthetics.

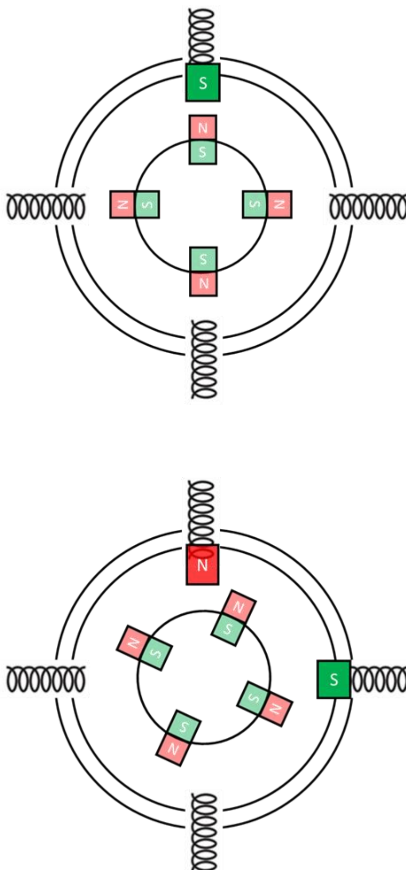


Figure 1.8 Stepper motor starting one clockwise step driven by magnetic attraction and repulsion

1.10 Electro-mechanics of a Stepper Motor

The motor shaft is connected to an armature which rotates; think of this as having small magnets attached. Surrounding the armature are coils which produce a magnetic field when driven by a current. Fig.1.8 shows a simple example. At the top the armature is held in its position by the south pole coil attracting the north pole on the armature. Then the coil south pole is rotated clockwise, and a coil north pole is placed at the top, so the armature will rotate 90 degrees then stop there. So, steppers work by rotating a magnetic field around the armature, dragging it around. Clearly the rotation occurs in steps defined by the number of magnets.

Our motors also have a gearbox which reduces the angle moved, this results in one revolution having 2038 steps. The angle of a single step (radians) is then

$$\Delta\theta_1 = \frac{2\pi}{2038}$$

which is about 0.0031 radians or 0.177 degrees, quite small. The distance covered by a wheel of radius r for one step is just

$$\Delta s_1 = r\Delta\theta_1$$

for our wheel of radius about 25mm this is around 0.08mm which is quite a small distance. This raises the hope of being able to control the position of a robot very accurately.

There are some limitations where this accuracy may not be realizes. One is related not to the motor, but to the wheels which may slip, especially if the motors are driven from rest to a high angular speed; to avoid this *speed ramping* will be used. Others are related to the motor, the motor may miss a step, especially when its speed is changed dramatically; ramping may help this. The other issue relates to driving two motors at the same time to make the robot move forward in a straight line. Assuming the following function is available (where the arguments are steps to the left and right motors respectively); compare the two coding solutions to get the robot to move *forwards* 100 steps.

`stepMotors(100,100);`	`int i=0;`
	`while(i<100) {`
	`    stepMotors(1,1);`
	`    i++;`
	`}`

The solution on the left will not work, since the internals of the function stepMotors(…) cannot drive both motors at the same time, and almost certainly drive one motor 100 steps then the other motor 100 steps. So, the robot would waddle forward in a series of arcs. While the same is true for the robot

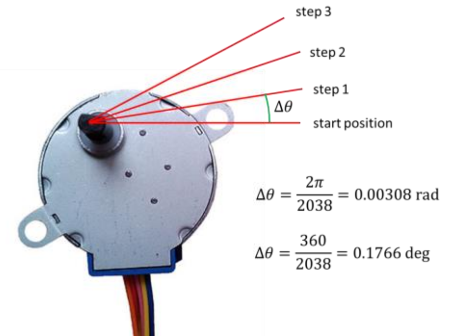


Figure 1.9 Our stepper showing a few anti-clockwise steps

on the right, the waddle is limited to single steps and will be barely noticeable.

1.11 Forward Motion in a Straight Line

This is straightforward; we must arrange two things, (i) both left and right motors take the same number of steps, (ii) we must drive them with the same speeds. The API has two functions

```
setStepperSpeeds(speedL,speedR);

stepMotors(nL,nR);
```

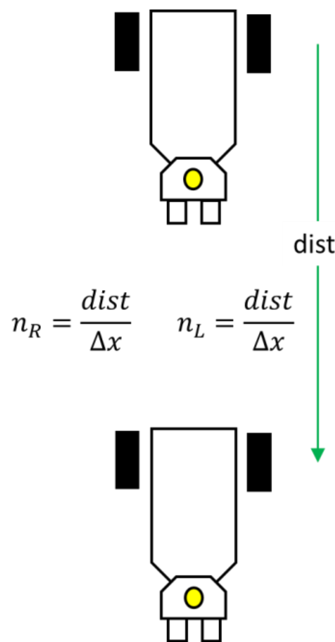


Figure 1.10 Motion on a straight line; same speeds, same number of steps.

where **nL** and **nR** are the numbers of steps sent to the left and right motors. The calculation of these is straightforward; if we want to drive the robot **dist** forward then we have

$$n_L = \frac{dist}{\Delta x}, \qquad n_R = \frac{dist}{\Delta x},$$

where Δx (**dx** in code) is the distance travelled for one motor step. Fig 1.10. summarizes this.

1.12 Motion on an Arc

This is a little more complicated to analyze, but the implementation of the algorithm in code is really tricky and requires some thought. Let's say we want to move the robot along an arc of radius R and angle $\Delta\theta_{rob}$ in a clockwise manner. We have seen something similar before; the arrangement is shown in Fig.1.11. The left wheel moves further than the right wheel, and since both wheels must finish their journeys *at the same time*, the left wheel must move proportionally faster.

The distances travelled (Fig.1.11 top) are calculated as usual; for the left wheel we have

$$\Delta s_L = (R + a)\Delta\theta_{Rob} \qquad (1)$$

and for the right wheel

$$\Delta s_R = (R - a)\Delta\theta_{Rob} \qquad (2)$$

Therefore, using the relation

$$n_L = \frac{\Delta s_L}{\Delta x} \qquad (3)$$

and an equivalent one for the right wheel, we find

$$n_L = \frac{\Delta\theta_{Rob}}{\Delta x}(R + a) \qquad (4)$$

and

$$n_L = \frac{\Delta\theta_{Rob}}{\Delta x}(R - a) \qquad (5)$$

Everything on the right side of these equations is known, so we can compute the numbers of steps needed by left and right motors.

As mentioned, both motors need to complete their rotations at the same time, let's call this time Δt. Since distance is velocity time we have

$$\Delta s_L = v_L\Delta t, \qquad \Delta s_R = v_R\Delta t$$

and dividing these

$$\frac{\Delta s_R}{\Delta s_L} = \frac{v_R}{v_L} \qquad (6)$$

and using expression (3) we have

$$\frac{v_R}{v_L} = \frac{n_R}{n_L} \qquad (7)$$

so, the speeds are in proportion to the number of steps taken, in this case the right motor has a lower speed, agreeing with Fig.1.11.

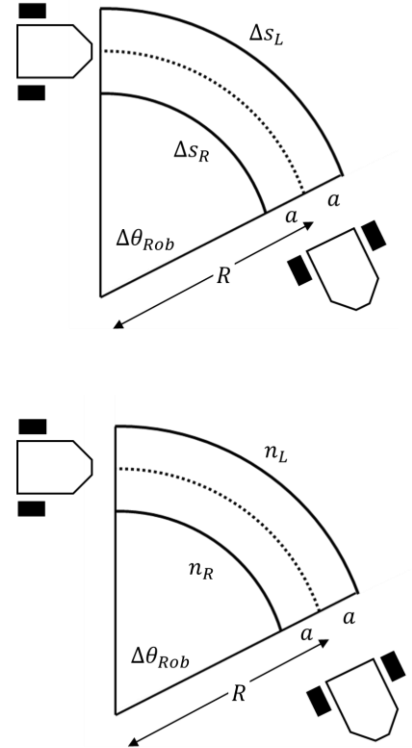


Figure 1.11 Driving on an arc: Top shows the distances, bottom expresses these in number of left and right steps.

We need to use expressions (4,5, and 7) in our code. These are used once to calculate the number of steps and the speeds of both motors. The problem is, once we have these values, how to use them to get the motors to turn correctly.

The approach we take (which is successful) is as follows. The right wheel moves a shorter total distance than the left wheel. So, when the right wheel has taken *one step,* we *imagine* that the right wheel has taken a *fractional step*. Of course, it can't but we imagine it can. When the left wheel takes another step, we imagine the right wheel as taken another fractional step.

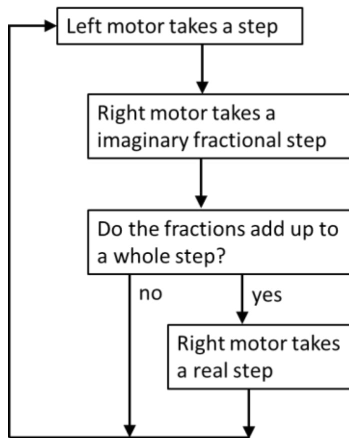


Figure 1.12 Arc algorithm

At some time, the right wheel fractional steps will add up to a whole step, so at this point we make the right wheel take a step. How do we calculate the size of this imaginary step? Well, it is simply the ratio

$$\beta = \frac{n_R}{n_L} \qquad (8)$$

so, when the left motor has finished its n_L steps, the total steps taken by the right motor is

$$\beta n_L = \frac{n_R}{n_L} n_L \qquad (9)$$

which is just n_R, exactly what we want! Fig.1.12 shows this algorithm expressed as a flow diagram.

One solution to coding this is shown below. This assumes required step numbers and speeds have been computed. The loop runs over the left steps, and the variable **prepR** ('prepare the right motor') accumulates the fractional imaginary right motor steps, **beta** since $n_L = 1$ in (8). When **prepR** is greater than one step, the right motor is stepped. Note when the loop over the left steps is finished, there is a little check to see if there is an outstanding step for the right motor.

```
while (doneL < nL) {  // --------

  stepMotors(1,0);
  doneL += 1;
  prepR += beta;

  if(prepR > 1.0) {
      stepMotors(0,1);
      doneR += 1;
      prepR -= 1.0;
  }

} // End while ----------------

// Check if any outstanding Right
if (prepR > 0.5) {
  stepMotors(0,1);
  doneR += 1.0;
  prepR -= 1.0;
}
```

The variables **doneL** and **doneR** count the steps actually taken while **nL**and **nR** refer to the total steps to take. This code will only work for clockwise turns, and needs to be generalized for all arcs.