Version 24-10-20

Chapter0 Introduction

This will probably be the last section I write!

Chapter 1 How to program the Controller

1.1 Overall structure of a Controller

1.2 Finite State Machine Model

1.3 Hybrid Model

~~Often we find ourselves needing states within states. For example, in a maze-solving algorithm there will be a state TURN_LEFT when the robot needs to turn left at a junction (Figure X). But this manoeuvre may need a few sub-states, such as moving to the centre of the corridor, then the actual turn left, then moving away from the junction until it is clear. Sure we can code this in an extra FSM, but we can do this implicitly.~~

~~figure~~

```
    while(dist < 0.040){
      wheelTheta = wb_position_sensor_get_value(leftWheelAngle) - wheelThetaStart;
      dist = wheelTheta*WHEEL_RADIUS;
      printf(">>> dist = %f\n",dist);
      omegaL = 0.1*MAX_SPEED;
      omegaR = 0.1*MAX_SPEED;
      wb_robot_step(TIME_STEP);
    }
```

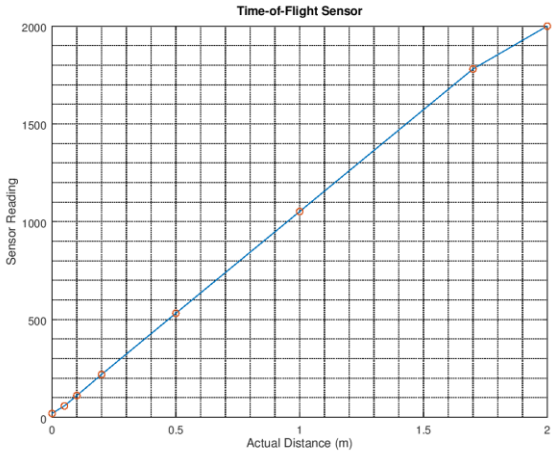
Chapter 2 Sensors

Sensors measure something in the *physical* world, and report a reading to the robot. A thermometer measures the temperature of the environment, and returns a number, degrees. A distance sensor will measure the distance of the robot to a physical obstacle, and report back some number. The relationship between the actual physical distance and the reported number can be quite unusual.

Here we discuss some of the sensors available on the ePuck-2 robot: (i) time-of-flight distance sensor, (ii) infra-red distance sensor (short range), (iii) light-level sensor, (iv) color camera.

2.1 Time of Flight Sensor

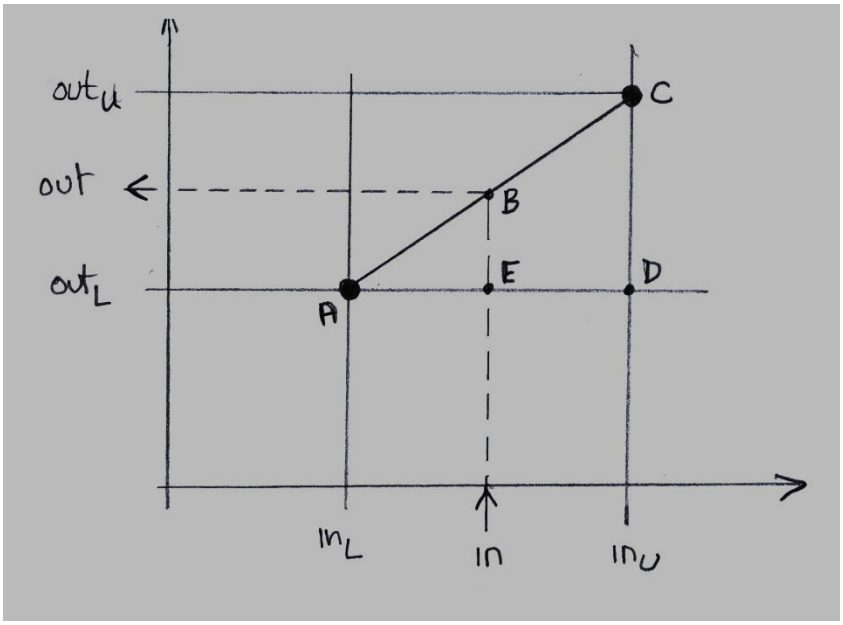
The relationship between distance and sensor reading is shown in the graph below. The relationship is almost linear. when the distance is 1m, the sensor reading is around 1100 and when the distance is 2m the sensor reading is 2000. Here you can see the sensor reading is about 1000x the actual distance.



Such a curve is established by actual measurements. These are shown as red circles in the above graph, and have been provided by the Webots ePuck-2 PROTO node. The experimental results are entered into a 'lookup table' (LUT) and here is the LUT for the time-of-flight sensor. Note that all sensor LUTs have this format. The first column is the physical distance in metres, the second is the sensor reading measured at that distance. The third column is an estimate of the experimental error at that distance, obtained from multiple readings. You can see agreement between the LUT and the above graph, where I have plotted column 2 versus column 1 (and have ignored the error numbers … for the moment).

```
lookupTable [
  0.00   19.8  0.126
  0.05   58.5  0.032
  0.10  111.0  0.019
  0.20  218.9  0.009
  0.50  531.9  0.007
  1.00 1052.0  0.010
  1.70 1780.5  0.013
  2.00 2000.0  0.000
]
```

So how does this work. Say the robot is 1.00 metres away from an obstacle, then the sensor will report 1052.0, and when the robot is 0.1 metres away from the obstacle, the sensor reports 111.0. But what about if the distance is not in the LUT, e.g., 0.25 m or 1.5m? Well we use a technique called ***linear interpolation*** and here is an explanation of that.



The values associated with the large dots are known. These are the LUT values. Our physical distance is *in* and the sensor reading we need to calculate is *out*.

The basic concept behind linear interpolation rests upon the fact that triangles ABE and ACD are *similar* which means that their angles are correspondingly equivalent. So we have

$$\frac{BE}{AE} = \frac{CD}{AD}$$

Therefore

$$\frac{out - out_L}{in - in_L} = \frac{out_U - out_L}{in_U - in_L}$$

Therefore

$$out = out_L + (in - in_L)\left(\frac{out_U - out_L}{in_U - in_L}\right)$$

So say our LUT looks like this

```
lookupTable = [
  3  100
  5  200
]
```

Then the above expression becomes

$$out = 100 + (in - 3)\left(\frac{200 - 100}{5 - 3}\right)$$

So if the physical distance is 4 then we have

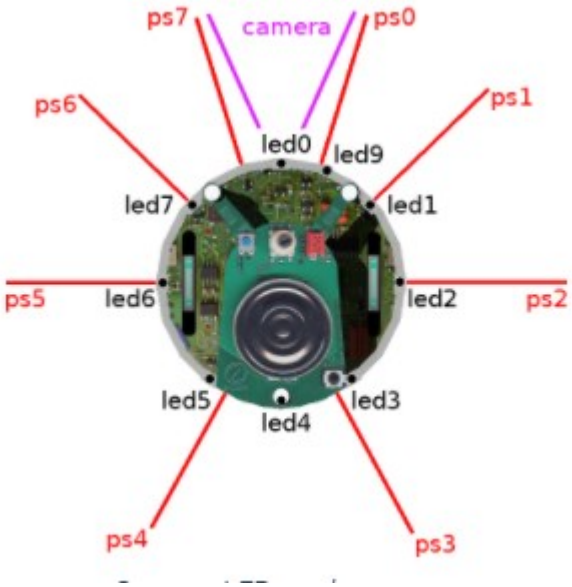
$$out = 100 + (4 - 3)(50)$$

This gives us a sensor reading of 150. Clearly that makes sense, since the value of 4 is half way between the two LUT input values, so the output should be half way between the output values. So that is linear interpolation.

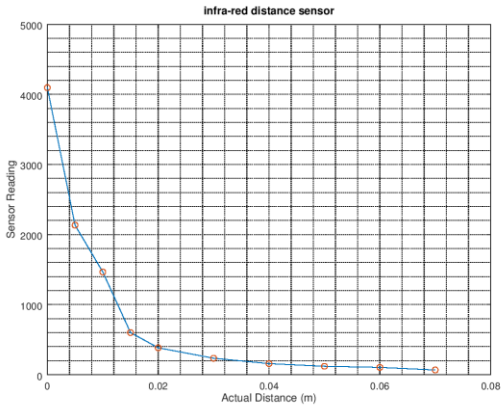
An example of how to code a LUT is provided in the code snippets section, together with linear interpolation.

2.2 Infra-red Distance Sensors. Al

Here's the arrangement and numbering of the IRed sensors. All sensors have a distance 3.25 cm from the robot centre and are located at the following angles: ± 17, 45, 90, 150 degrees.



Here's the graph for the infra-red sensors on the ePuck-2. The first thing to note is that they are extremely short range, and detect only up to 0.07 metres (7cms), so these sensors are useful for collision detection and avoidance, not for target localization. The second thing to note is that the sensor readings get smaller as the physical distance increases. This is rather inconvenient, so we can code up our own 'inverse LUT' which takes in sensor readings and outputs the corresponding physical distance. How to do this is shown in the code snippets.



2.4 Light Sensor

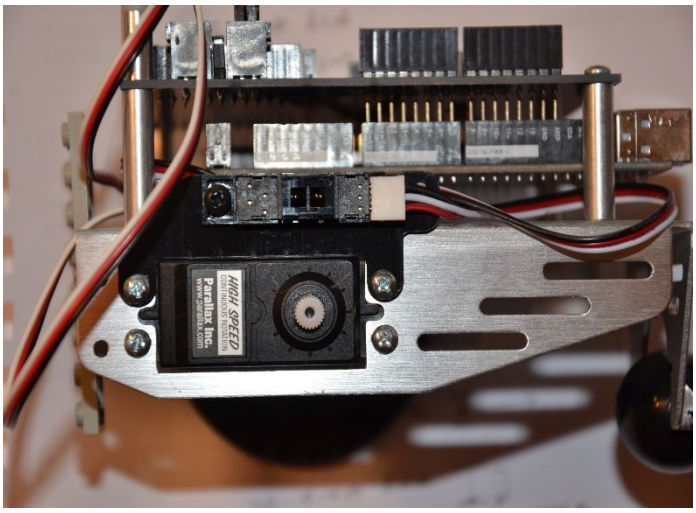
Here the LUT is rather simple

```
lookupTable = [
  1  4095
  2  0
]
```

These two data points lie on a descending straight line. This is easily inverted using a simple maths function which can be implemented in a single line of code.

2.5 Wheel Encoders (Odometry)

These are on-board sensors that measure the *actual* rotation of the robot wheels. A typical arrangement, used by the Boebot is shown in the figure below.



On the left you can see a wheel with a ring of slots. On the right you can see the motor connected to the wheel (removed). Just above the robot, facing you there is an infra-red transmitter and receiver (black, red, white and brown wires connected). So when the wheel rotates, the infra-red beam either passes through the slots or reflects from the plastic between the slots. The receiver then gets a pulse of infra-red each time a slot passes by. By counting the pulses, the receiver knows the actual angle turned by the wheel.

Since the robot is aware of the angle of its wheels at any time, it can calculate the distance it 'thinks' it has moved. This is unlikely to be the actual distance, since wheels may slip, especially when they accelerate from zero speed to a large speed. Therefore the distance the robot thinks it has moved is likely to be an over-estimate.

Now let's do some calculations using these distances, and see what we can find out about possible robot trajectories. The equations presented below can be applied to perceived or actual distances, their interpretation will depend on whether we are thinking actual or perceived distances.

First, we can calculate the distance moved by a wheel from the measured encoder angle.

$$s = R\theta \qquad (2.1)$$

Now the distances travelled by left and right wheels,

$$s_L = (R - a)\theta_{Rob} \qquad (2.2)$$

$$s_R = (R + a)\theta_{Rob} \qquad (2.3)$$

Subtracting 2.2 from 2.3, and solving for *R* gives us

$$\theta_{Rob} = \frac{s_R - s_L}{2a} \qquad (2.4)$$

This could be useful, since if we can measure the distances travelled by the wheels, we can calculate the angle of rotation of the robot. By substituting 2.1 into 2.4, we can also calculate the angle of rotation based on the encoder values

$$\theta_{Rob} = R\frac{\theta_R - \theta_L}{2a} \qquad (2.5)$$

Using the encoder values gives us the *perceived* rotation of the robot, what angle the robot thinks it has turned.

Now if we add 2.2 and 2.3 we get

$$R = \frac{s_R + s_L}{2\theta_{Rob}} \qquad (2.6)$$

and finally substituting 2.5 into 2.6 to get rid of θ_{Rob} we have the result

$$R = \frac{s_R + s_L}{s_R - s_L} a \qquad (2.7)$$

We can also express this in terms of encoder angles by using 2.1

$$R = \frac{\theta_R + \theta_L}{\theta_R - \theta_L} a \qquad (2.8)$$

This is very useful since it tells us the radius of curvature of the robots path based on the encoder readings. We can make this even simpler by considering the *ratio* of the left and right distances or encoder readings.

Let us say that the right turns faster than the left wheel (and therefore moves further) then we can define a ratio α such that

$$s_R = \alpha s_L \qquad (2.9)$$

Then finally we get

$$R = \frac{(\alpha + 1)}{(\alpha - 1)} a \qquad (2.10)$$

Let's take some special cases:

(1) $\alpha = 1$ which means $s_R = s_L$, that is the wheels are rotating with the same speed. The bottom of (2.10) is zero, so that *R* becomes infinite. So the robot travels round a circle of infinite radius, which means it goes in a straight line.

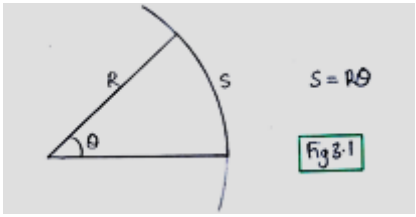
(2) $\alpha = -1$ which means $s_R = -s_L$, that is the wheels are rotating at the same speed, but in opposite directions. The top of (2.10) is zero which means the circle has zero radius *R* in other words the robot rotates about its centre. If we assume that $s_L > 0$, i.e. the left wheel is moving forward, then the right wheel is moving backwards, so the robot rotates in a clockwise sense.

(3) α is very large (much greater than 1). This means the right wheel is rotating much faster than the left wheel which is hardly rotating. Then, in 2.10 the +1 and -1 can be forgotten about and we have α divided by α which is 1, so we have *R = a*. This means the robot will rotate about its left wheel in a clockwise direction.

Chapter 3 Robot Kinematics

Basic Theory

The only way to get a wheeled robot to move is to drive its motors to turn its wheels. When the wheels turn, each wheel moves itself forwards. We *drive* each wheel by setting its angular velocity. In code, for the left and right wheels, these angular velocities are **omegaL** and **omegaR.** The question is how do we set these values to make the robot move as we want, e.g., to a point (a target) or along a curve (to avoid an obstacle). To answer this, we must understand how the robot moves when we drive the motors, this is called *robot kinematics.*

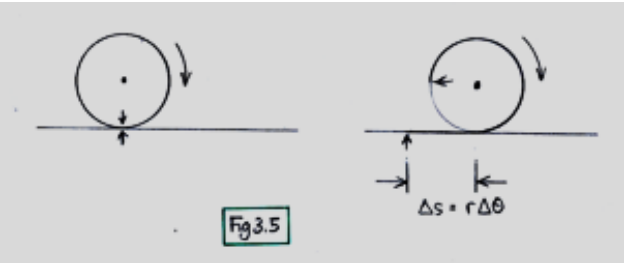


The basic maths needed to understand all of the following is shown in Figure 3.1 which expresses the relationship between how far we move around a circle of radius *R.* The key relationship between arc length *s* the circle radius *R* and the angle θ is

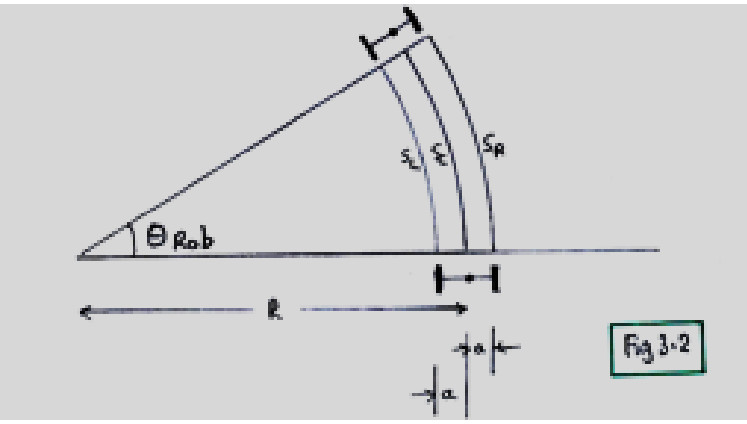
$$s = R\theta \qquad (3.1)$$

First let's apply this to a single wheel rotating. As you can see in Fig. 3.5 when the wheel rotates the distance travelled is 'unrolled' from its circumference. This is simply

$$s_L = R\theta_L \qquad (3.1a)$$



Now let's apply this to a 2-wheeled robot moving around a circular arc. (Fig. 3.2).



Here the distance *a* is half the distance between the wheels. Clearly the right wheel travels further along a circle with a larger radius than the left wheel. So, using the above formula 3.1, we have for the left and right wheels, where θ_{Rob} is the angle turned by the robot (i.e., not its wheels)

$$s_L = (R - a)\theta_{Rob} \qquad (3.2)$$

$$s_R = (R + a)\theta_{Rob} \qquad (3.3)$$

Also we have for the centre of the robot

$$s_C = R\theta_{Rob} \qquad (3.4)$$

Now if we add the equations for s_L and s_R we get

$$s_L + s_R = 2R\theta_{Rob} \qquad (3.5)$$

and substituting $R\theta_{Rob}$ from 3.4 we find

$$s_C = \frac{(s_L + s_R)}{2} \qquad (3.6)$$

which tells us that the distance travelled by the robot is the average of the distances travelled by the two wheels.

Now we can think about how to calculate the motor drive angular speeds. The notation for these, in maths and in code is ω_L **omegaL** and i ω_R **omegaR**. But first we need to think about what *speed* actually means. You will be familiar with things moving in straight lines, with a speed in metres per second. To calculate the speed of say a car, you measure the distance gone, Δx in a certain time interval Δt and you calculate the speed like this

$$v = \frac{\Delta x}{\Delta t} \qquad (3.7)$$

For circular motion, the expression is the same, but in place of a change in distance, we use the change in angle like this

$$\omega = \frac{\Delta \theta}{\Delta t} \qquad (3.8)$$

So we take 3.2

$$s_L = (R - a)\theta_{Rob} \qquad (3.9)$$

and, using 3.1, convert s_L the distance moved by the left wheel to θ_L, the angle moved by the left wheel so we get

$$r\theta_L = (R - a)\theta_{Rob} \qquad (3.10)$$

Now let's say the left wheel has rotated angle θ_L in time Δt, then its angular velocity is $\omega_L = \theta_L/\Delta t$, so inverting this we have $\theta_L = \omega_L \Delta t$ which we substitute on the left hand side of the above expression. Similarly we have for the robot angle $\theta_{Rob} = \Omega_{Rob}\Delta t$, where Ω_{Rob} is the angular velocity of the whole robot, as it turns. So, doing the substitutions and dividing by *r we* have

$$\omega_L = \frac{(R - a)\Omega_{Rob}}{r} \qquad (3.11)$$

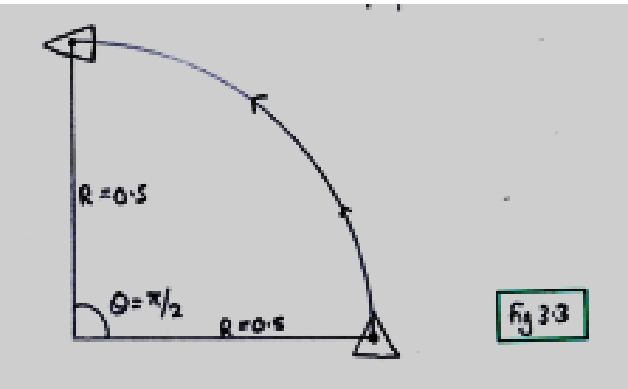
and similarly for the right wheel

$$\omega_R = \frac{(R + a)\Omega_{Rob}}{r} \qquad (3.12)$$

These are important equations, so let's see what they tell us. The left-hand sides give the 'outputs' of the equations, in other words we get **omegaL**, and **omegaR** which we need to drive the motors! Also, we know the value of wheel radius *r* and of the axle half-length *a*. So we have two variables 'inputs' which we can set to get the robot moving around a circle of radius *R* with a particular speed of rotation Ω_{Rob}.

Motion on a Circular Arc

Let's take a practical example, a robot moving around a circle of radius 0.5m for an angle of $\pi/2$ (90 degrees). This is sketched in Fig. 3.3.



Let's say we want the robot to take T=10 seconds to do this. Then its angular speed must be

$$\Omega_{Rob} = \frac{\Delta\theta}{\Delta t} = \frac{\pi/2}{10} = 0.157 \qquad (3.13)$$

Then the motor drive angular velocities become

$$\omega_L = \frac{(0.5 - 0.0205)}{0.053/2} 0.157 = 2.841 \qquad (3.14)$$

$$\omega_R = \frac{(0.5 + 0.0205)}{0.053/2} 0.157 = 3.084 \qquad (3.15)$$

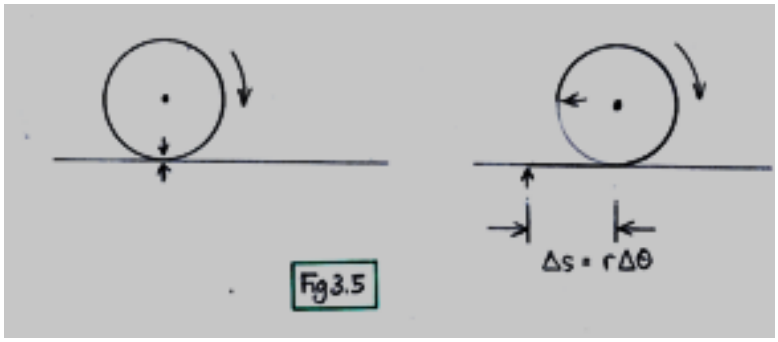
So the robot will travel around ¼ of a circle of radius 0.5 metres in 10 seconds. Finally, let's make these expressions general, to get the robot to go around a circle or radius R for desired angle θ_{des} in a desired time T_{des}

$$\omega_L = \frac{(R - a)}{r} \frac{\theta_{des}}{T_{des}} \qquad (3.16)$$

$$\omega_R = \frac{(R + a)}{r} \frac{\theta_{des}}{T_{des}} \qquad (3.17)$$

Motion on a Straight line

Let's take another important example, a robot moving on a straight line. Here its angular velocity is zero, since it is not turning. Both wheels travel at the same velocity. As you can see in Fig. 3.5, as the wheel rotates, the distance travelled is the distance 'unrolled' from the wheel's circumference.



This is simply $\Delta s = r\Delta\theta$, so that if the desired forward velocity is V_{Rob} then

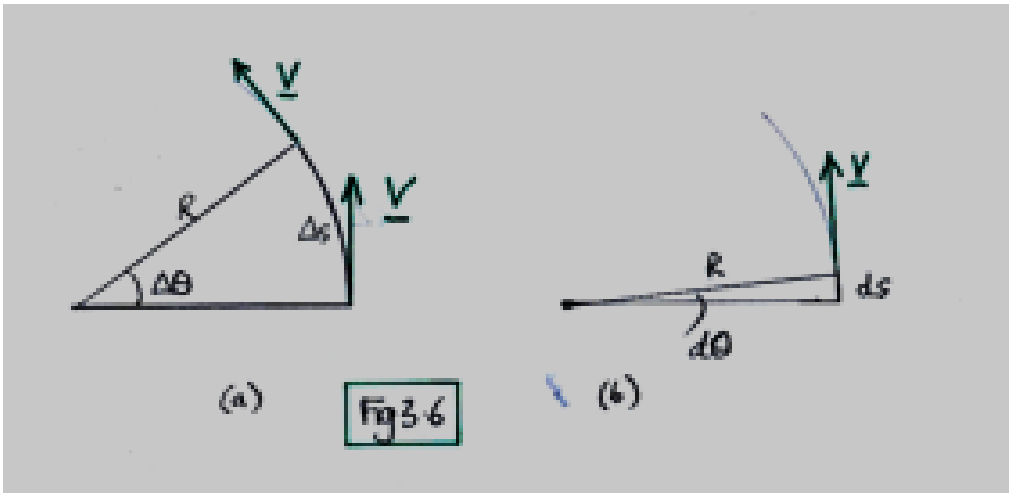
$$V_{Rob} = \frac{\Delta s}{\Delta t} = \frac{r\Delta\theta}{\Delta t} = r\omega_L$$

hence we can set the omegas,

$$\omega_L = \omega_L = \frac{1}{r}V_{Rob} \qquad (3.18)$$

Generalized Motion

So we have seen that the motion of the robot body can be in a straight line with linear velocity V_{Rob}, or it can be a rotation of the robot about its centre with angular velocity Ω_{Rob}. The next question is can a robot have a mix of linear and angular velocities, and if so (and the answer is "yes") what does this mean?



Look at Fig. 3.6. On the left we have our standard diagram and we can write

$$\Delta s = R\Delta\theta \qquad (3.19)$$

therefore

$$\frac{\Delta s}{\Delta t} = R\frac{\Delta\theta}{\Delta t} = R\Omega_{Rob} \qquad (3.20)$$

hence

$$\Omega_{Rob} = \frac{\Delta\theta}{\Delta t} \qquad (3.20a)$$

Now $\Delta s/\Delta t$ looks like a velocity since it is a distance divided by a time, but the problem is, it is not measured in a straight line, so it cannot be the robot's *linear* velocity V_{Rob}. Fig. 3.6(a) clearly shows this, the robot has changed direction, so the velocity (shown as a vector) is changing with time. But if we make $\Delta\theta$ smaller and smaller, we will arrive at Fig. 3.6(b) where the angle change is *infinitesimally* small. Therefore the change in velocity is *infintessimally* small, so it does not change at all. So then we can define V_{Rob} like this

$$V_{Rob} = \frac{ds}{dt} = R\frac{d\theta}{dt} = R\Omega_{Rob} \qquad (3.21)$$

that is

$$V_{Rob} = R\Omega_{Rob} \qquad (3.22)$$

Now we shall consider the situation if we want the robot to move with a desired linear velocity V_{Rob}, and a desired angular velocity Ω_{Rob}, how to we calculate the wheel rotation speeds ω_L and ω_R, since we drive the robot by specifying these speeds. Well we need to combine equations 3.22 with equations 3.11 and 3.12. Let's look at 3.11 in detail. First expand all terms,

$$\omega_L = \frac{R\Omega_{Rob} - a\Omega_{Rob}}{r} \qquad (3.23)$$

Then substitute 3.22 in the first term on the right, and then divide both sides by *r* to give

$$\omega_L = \frac{V_{Rob} - a\Omega_{Rob}}{r} \qquad (3.24)$$

And the same for 3.12

$$\omega_R = \frac{V_{Rob} + a\Omega_{Rob}}{r} \qquad (3.25)$$

These equations tell us for desired V_{Rob} and Ω_{Rob} (which we plug in on the right) we now know what to send to the motors, **omegaL** and **omegaR** (which pops out of the left).

There is one final bit of work to do. Since we know the robot's velocities, we should be able to calculate their location (x,y) coordinates in space. The thing to remember is that the robot's velocity most likely will change direction in time. Look at Fig. 3.7. This depicts a robot moving along a curve (this is not a simple circular arc). The robot is shown at a particular time t_1 at location (x_1, y_1) and it makes an angle θ with the x-axis. The velocity *V* is tangential to the curve, and this can be decomposed into two components. $V_x = V\cos\theta$ and $V_y = V\sin\theta$. Therefore in an infinitesimally small time interval Δt then the robot will move from (x_1, y_1) to a new location

$$x_2 = x_1 + V_{Rob}\cos\theta\ \Delta t$$
$$y_2 = y_1 + V_{Rob}\sin\theta\ \Delta t \qquad (3.26)$$

with

$$\theta_2 = \theta_1 + \Omega_{Rob}\Delta t$$

where here the Δt s are infintessimal.

So to calculate the location of the robot at any time, we need to sum up a series of little moves in the x and in the y directions. Mathematically, this process is called *integration.*

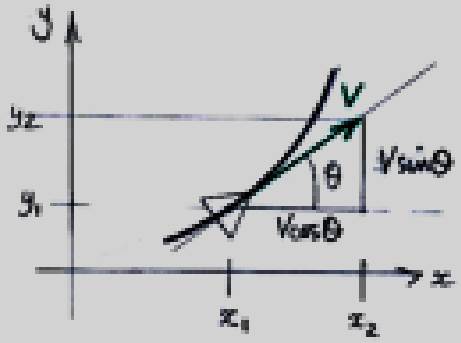
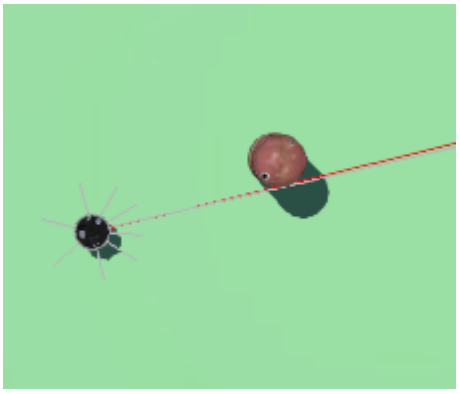


Fig 3-7

Chapter 4 Object Detection and Localization

4.1 Localization by Vector Field Histogram

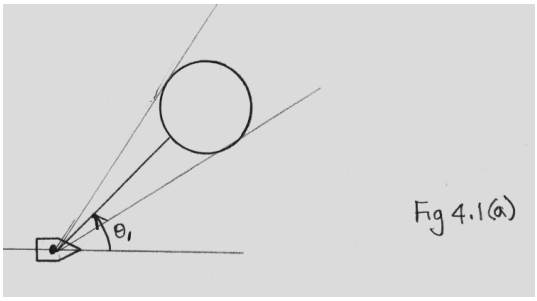
Here's a robot scanning the arena by rotating in the positive anti-clockwise direction. Its time-of-flight sensor ray has just started to intersect the object.



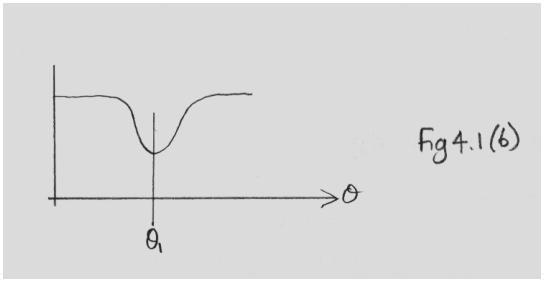
As the robot rotates full circle, the distance to any intersecting object is measured and this is stored in a distance array:

```
double sensorVal = wb_distance_sensor_get_value(toflight);
double dist = lookUpDistanceTOF(cTOF,sensorVal);
distArray[thetaIndex] = dist;
```

Note the index into the array **thetaIndex** is the angle theta at which the ray is pointing as shown below

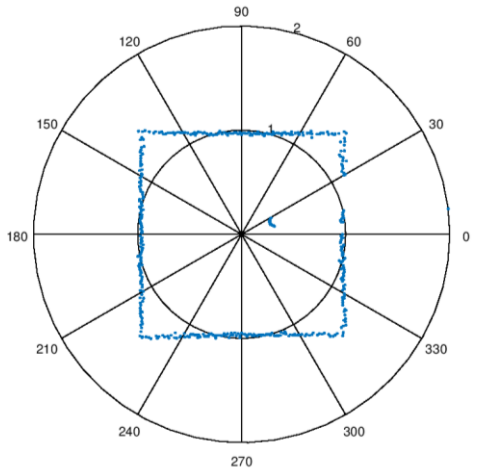


When the scanning is complete, the 'histogram' of distances versus scanning angle look like this where angle theta is along the bottom and distance is up the side.



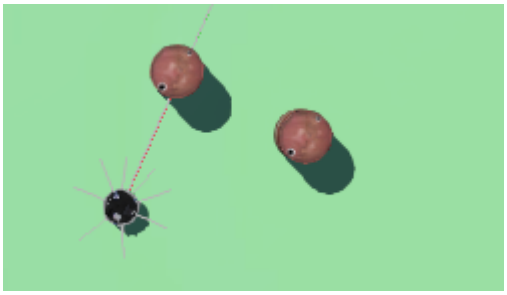
You can see that most of the arena is far from the robot, but that the obstacle is seen as a clear dip in the distance histogram. It is a simple matter of finding this dip by looking for the minimum distance. This minimum gives (i) the distance to the obstacle's nearest point, (ii) the angle θ_1 of this point. In other words, the obstacle is localized in space.

Below is a polar plot of distance for the complete circle. You can clearly see the arena walls and also the obstacle barrel.

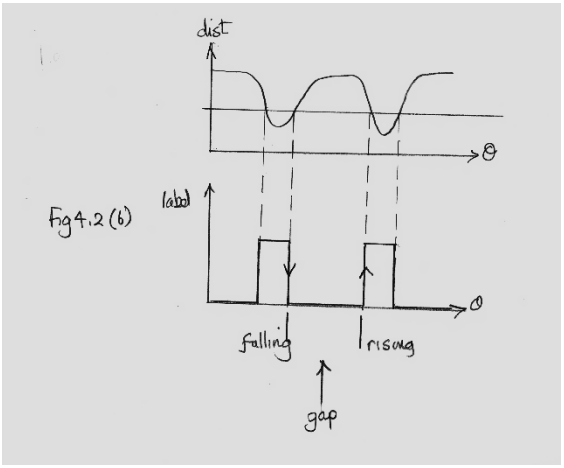


4.2 Locating gaps between obstacles

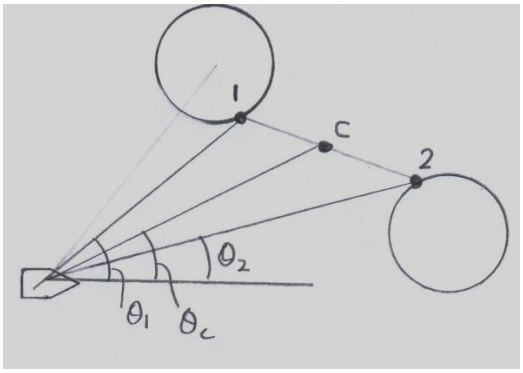
Here we have a similar situation, but we assume there are two objects, and the robot needs to navigate through the gap between them



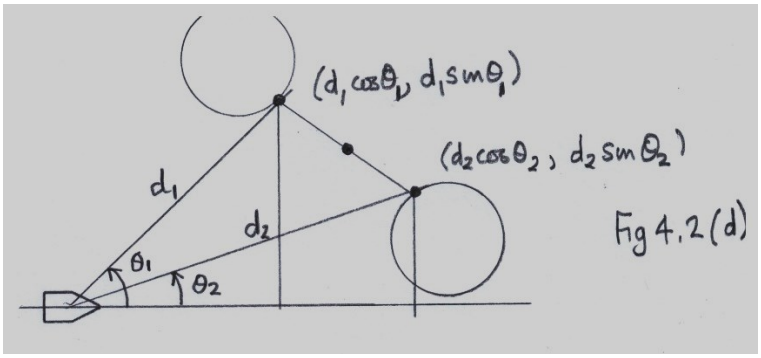
The robot rotates as in 4.1 and builds up its histogram which now contains two dips as shown below



Now we need to locate the gap which corresponds to an angle mid-way between the two dips in the distance histogram. This is not a trivial process due to the noise inherent in the distance measurement. We present one approach here. First a threshold is chosen, and all the distance points which are closer than this threshold are labelled '1' while the remaining points are labelled '0. These labels are stored in a second array **thetaArray.** Then this array is scanned from 0 degrees up to 360 degrees (left to right in Fig. 4.2(b)) and the falling and rising edges of the values in thetaArray are found. The angles of these two edges correspond to the part of the object closest to the gap. Finally, we take the angles corresponding to these two edges and average them to get the angle of the gap from the robot. This is shown below in Fig 4.2(c)



Finally the points of intersection with the obstacles can be found using straightforward trigonometry shown in Fig 4.2(d). The distances to the obstacles are taken from the **distArray** using the index of the falling and rising edges.

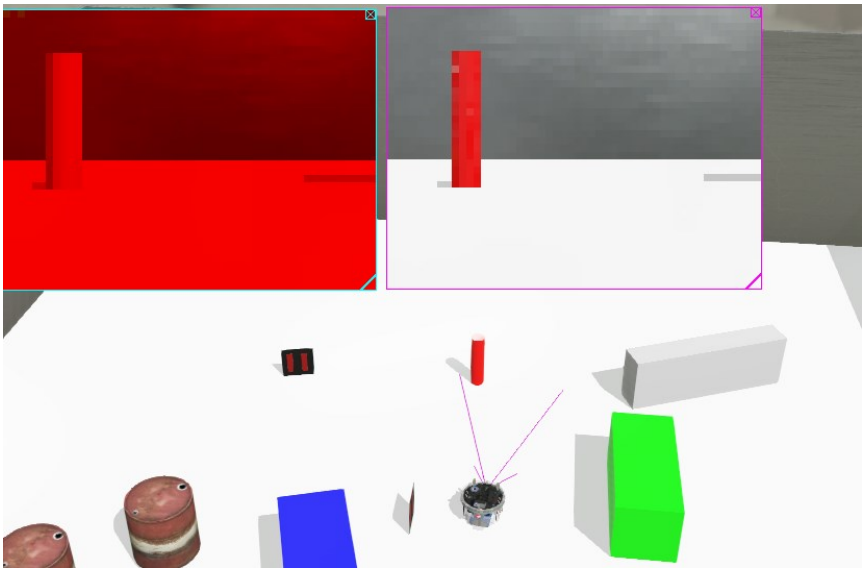


You may question the use of an 'arbitrary' distance threshold. But just look at the polar plot presented above. It should be, in most cases, straightforward to choose a suitable threshold.

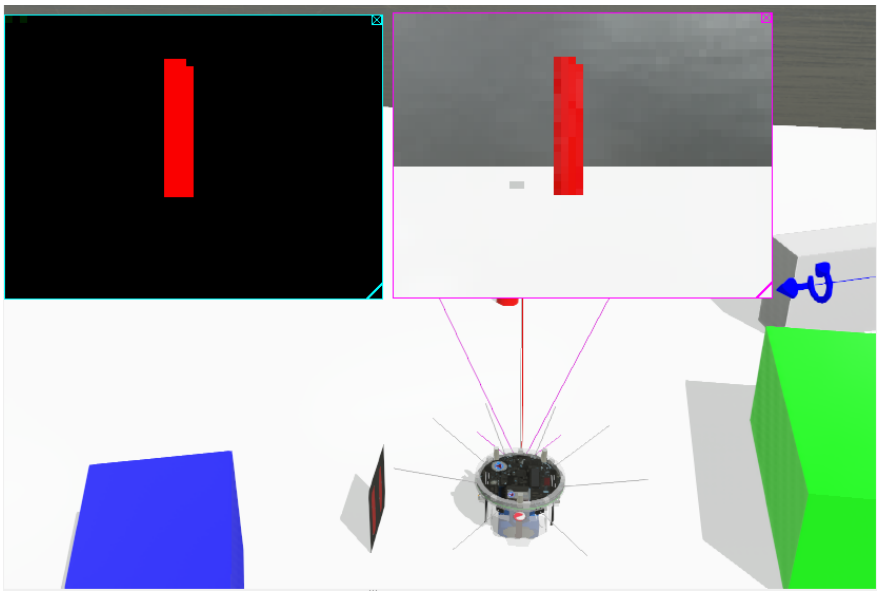
4.3 Computer Vision – Basics

Vision as a sensory input is truly amazing. We can recognize faces, read words and music, watch movies, walk around avoiding obstacles, climbing stairs and getting to our destination, driving, flying, astronauting, and much more all in real-human-time, (and often when parallel-processing, with our smart-phones). Yikes! So what is amazing is the amount of information we process in our brain and the speed of this. Remember our brain's neurons work at a speed of around 10 cycles per second; compare this with a CPU operating at over 1000 million cycles per second! And we are not the only living creatures able to do this; I observe my cats. So there must be some 'tricks' involved in reducing the amount of information we need to process in order to interpret our world as represented to us by our vision.

Let's look at how images are captured and represented by the ePuck robot. The on-board CCD camera provides us with a color image expressed in three channels, Red, Green and Blue. Here's a screen-shot of the ePuck looking at a scene containing a red cylinder. This shows the robot world with the camera RGB view top-right and the red-channel of this image top-left. What we are trying to do here, is to identify and localize the red cylinder, clearly visible in the RGB image on the right. But the red-channel of the RGB image does not help us, since the whole image has a substantial red component. The red cylinder has about the same R-values as the white floor (which has high values of R, G and B, since it is white).



We can solve this problem by converting the RGB image into an HSI image (Hue, Saturation, Intensity), and use the Hue component to segment the red object. Here's the result which clearly shows the red object has been segmented



The code to effect the transformation to Hue is straightforward. First we get the **red, green, blue** values from the camera for each pixel at column **i** and row **j**

```
double red = wb_camera_image_get_red(image, width, i, j);
double green = wb_camera_image_get_green(image, width, i, j);
double blue = wb_camera_image_get_blue(image, width, i, j);
```

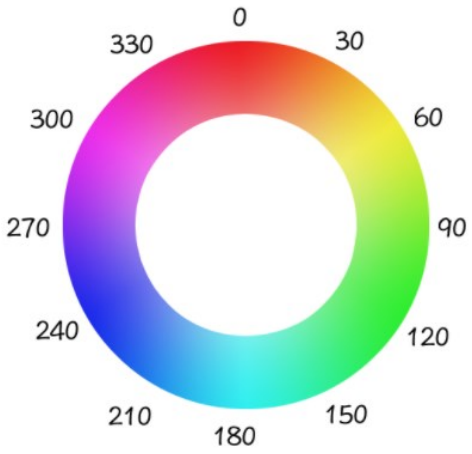
These values lie in the range 0 – 255, so we normalize them into the range 0.0 – 1.0

```
red = red/sum;
green = green/sum;
blue = blue/sum;
```

and finally calculate the Hue, where **acos** means the inverse cosine function

```
double numi=0.5*((red - green)+(red - blue));
double denom=sqrt((red-green)*(red - green)+ (red-blue)*(green - blue));
double hue = acos(numi/(denom + 0.0000001))*180/M_PI;
if(blue > green) hue = 360-hue;
```

The resulting Hue is an angle (sorry, angles again) on the color wheel



So if we want to label a pixel as red, then we need the following condition which will set a pixel in the thresholded image **imageThresh** to 250 if the Hue of the pixel in the source image is in range 330 to 40 degrees.

```
if(( hue > 0 && hue < 40)  || (hue > 330 && hue < 360)){
  *(imageThresh + i*4 + 1 + j*width*4) = 250;
}
```

Sounds simple, but there are some (hidden) problems. First if you look at the above code to transform RGB to Hue you will notice that the RGB values have been 'normalized' into a range 0.0 – 1.0, by dividing by the sum. This is necessary for the computation using the inverse cosine. But a pixel (255,255,0) will be normalized to (0.5,0.5,0.0), great. But a pixel (7,7,0) will also be normalized to (0.5,0.5,0). Both will be processed in the same way by the RGB to Hue computation and will be assigned the same hue. But the (7,7,0) pixel is likely to originate from noise in the camera (or elsewhere) while the (255,255,0) pixel is genuine data. We must reject the noisy (7,7,0) pixel and not use intensities below a certain value.

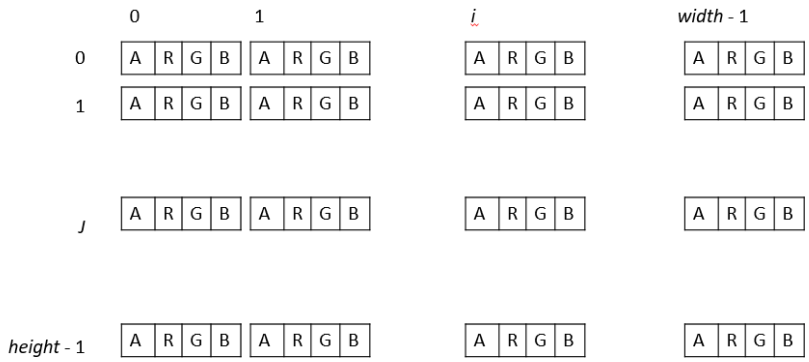
The second issue is about the *purity* of the color. The color wheel presented above assumes the Hue is *pure* in the sense that it is a combination of varying amounts of just two channels of R,G,B. So (139,200,0) and (0,111,218) are pure whereas (112,193,201) is not pure it is corrupted by some white light. Consider this pixel (R,G,B) = (250,150,50). We can write this as (250,150,50) = (200,100,0) + (50,50,50). The first part (200,100,0) is a pure colour since it has

just two RGB components. It is corrupted by a white level (50,50,50). So we have to remove any white level corruption. This can be done by subtracting the minimum of the RGB values for each pixel (here 50).

Now we must turn to the structure of an image from the ePuck camera, and how to manipulate this in our programs. This uses the OpenGL format where each image pixel is stored as 4 bytes as follows

Alpha	Red	Green	Blue

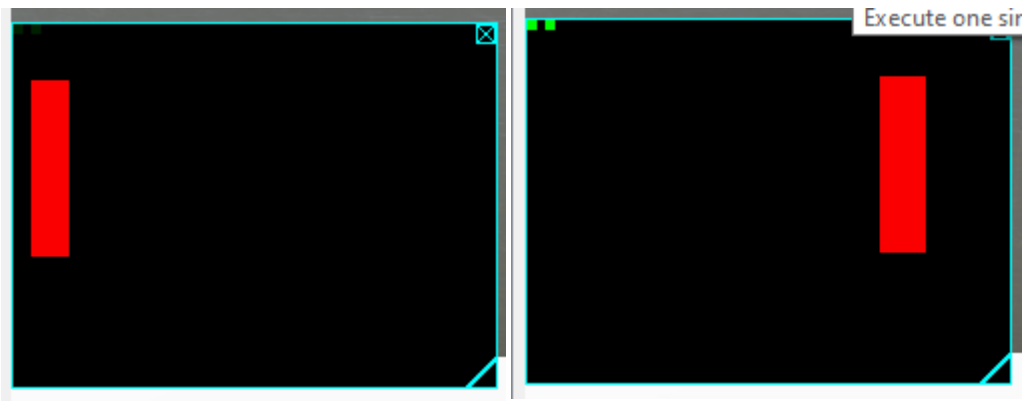
so the entire image which has *width* and *height* is represented as a 2D array of bytes like this, where *I* refers to the column in the array, and *j* refers to the row in the array. To get to the red value in the first row (*j = 0*) at any pixel *I* we must multiply *I* by 4 to get at the start of the 4 bytes for the pixel, then add 1 to get at the red byte. This is the term **i*4 + 1** in the above code. Then to drop down to the red pixel in the row below, we need to add on the number of bytes in each row. For row ***j*** this is just **j*width*4**.



Now to the expression ***(imageThresh + i*4 + 1 + j*width*4) = 250;** which sets the pixel at *row = j, col = I* to 250. Perhaps you were not expecting this, since the image is a 2D array. Perhaps you were expecting **imageThresh[row][col];** or something similar. But the truth is, we may have in our mind a 2D array model, but in reality the computer only has a contiguous 1D amount of memory, it has no concept of arrays at all! Now when we reference an array in memory, we need the **startAddress** where the array is and the **index** into the array. In C using 'pointers' we write this as ***(startAddress + index)** so now you should understand what the expression ***(imageThresh + i*4 + 1 + j*width*4) = 250;** is doing.

4.4 Computer Vision – Finding Objects

This is straightforward. Let us assume the objects are red, and we have a red segmented image where the pixels show where the scene contains only the red hue. As the robot is moving around, the camera may provide us with either of the images below, in the extreme cases.

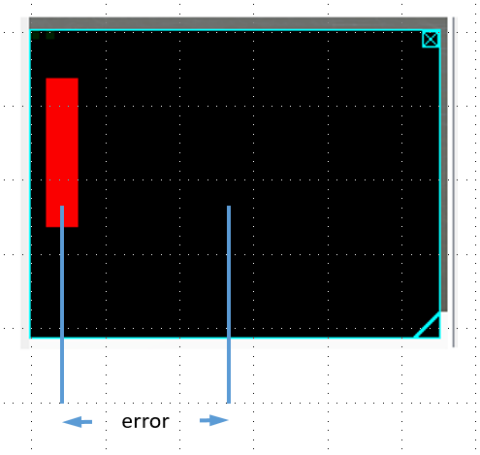


If we get the left image, then the robot needs to rotate anti-clockwise to bring the red blob towards the centre, and if we get the right image, the robot needs to rotate clockwise. So we need a function

int centre = getAvRedLocation(image1,width,height);

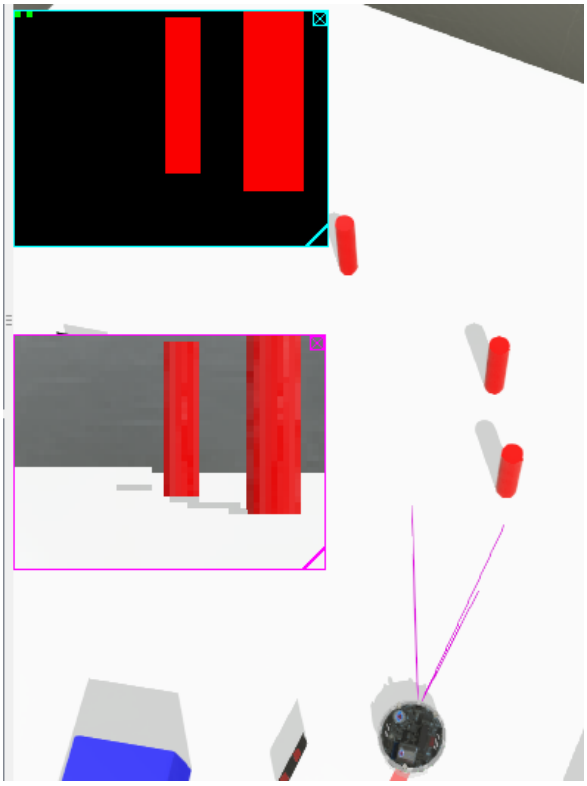
to return the centre column of the red blob, and depending on whether this **centre** is less or greater than the image centre (which is **width/2**) we turn in the appropriate direction. To make the computation more efficient, we do not process the entire image, but use a single row-slice through the image half way down the image, at **height/2**.

So, to make the robot rotate to the object, we define an *error* signal proportional to the difference between the centre of the object and the centre of the screen, and use this to drive the robot (in the correct direction).



4.5 Computer Vision – Navigating between Objects

This situation is a little more complex, since we have to deal with more than one object in the image. But we can introduce some order into the situation if all objects are the same size. So in the picture below the robot is presented with three red cylinders of identical radius and height. This means that the close the cylinder is to the robot, the wider and taller it will appear. The algorithm we shall use calculates the location of the *tallest* object in the image, and uses this to make the robot rotate away from it, since this is the closest to the robot.

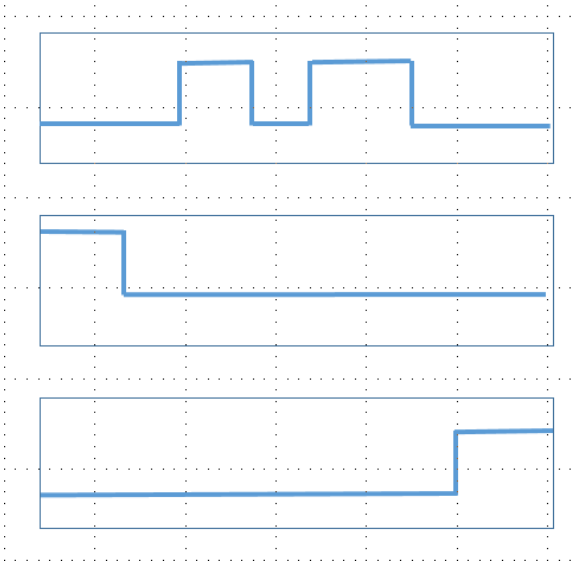


The camera frustrum (field of view) is shown on the robot, and at this time the robot can see the two rightmost columns. The closest is larger and the centre column of this object is used to signal the robot to rotate anti-clockwise to move away from the object.

The algorithm does not process the entire image, but to obtain efficiency takes a single row through the image at a value **height/2**. The algorithm scans this 1D array of values and identifies all objects it finds and stores their centre location in an array. Then it looks at the stored objects and calculates their heights by summing the number of red pixels in the central image column for the object. The algorithm then finds the maximum height and returns this to the main program which uses this height to control the robot's movement.

The code logs an object by looking at when the values in a row change from 250 to 0, the 'falling edge' of the object.

Here are three limiting types of row-scan through the image. The algorithm (code) must be able to handle all of these cases



Case 1: Two (or more) objects completely in the camera's field of view.

Case 2: Object partially captured at the left field of the camera

Case 3: Object partially captured at the right field of view of the camera

In most of the controllers seen so far, it is the angular velocities **omegaL omegaR** which we have changed directly to get the robot to move. This is *kinematic* control which means we change the robot speed directly.

In this case we use *dynamic* control. Here we change the angular acceleration of the robot body which then changes the angular velocity of the robot body which we use to set the wheel speeds using equations 3.11 and 3.12. So when the robot sees the tallest obstacle, it's rotational velocity is changed to turn it away from the obstacle, and it keeps on turning.

In other words, it has a short-term memory of having seen the obstacle, and continues rotating a little even when the obstacle moves out of sight. This is how real vehicles work, which maintain their momentum while moving.

[More on kinetic vs. dynamic control somewhere, sometime].

Chapter 5 Control of Robot Motion

Open and Closed-Loop control

Point and Shoot

Straights and Curves

Following a Parameterized Curve

Movement to a Posture

Chapter 6

Navigation

Bug-2 Wall-hugging

Potential (Force) Fields

Bug-2 Wall-hugging

Appendix A C-language basics

Appendix B Code Snippets

These snippets have come from my initial investigations of the Webots platform. They are non-trivial functions which you may use as part of your session work or mini-projects

---

(1) Converting a compass bearing to a 'maths angle'

```
bearingRobot = getBearing(comp);
theta = 180.0/2.0 - bearingRobot;
if(theta < -180.0)
   theta += 360;
```

---

(2) Function to calculate a bearing from the raw compass values

```
double getBearing(WbDeviceTag comp ){
   double bearing;
   const double *val = wb_compass_get_values(comp);
   double rad = atan2(val[0],val[2]);
   bearing = (180.0*rad)/M_PI - 180.0;
   if(bearing < 0.0)
      bearing = bearing + 360.0;
   return bearing;
}
```

---

(3) Function and LUT to get a distance in metres from an Infra-red raw sensor reading

This is C-syntax to declare a new data type.

```
typedef struct { double x; double y; }  LUT;
```

Here follows the 'inverted' LUT with sensor readings as inputs and distances as outputs

```
LUT c[10] = {
   {67.19, 0.07},
   {104.09, 0.06},
   {120.0, 0.05},
   {158.03, 0.04},
   {234.93, 0.03},
   {383.84, 0.02},
   {601.46, 0.015},
   {1465.63, 0.01},
   {2133.33, 0.005},
   {4095.00, 0.00}
};
```

Here is the linear interpolation computation

```
double lookUpDistance(LUT *c, double x) {
   int i;
   int n = 10;
   for( i = 0; i < n-1; i++ ) {
      if ( c[i].x <= x && c[i+1].x >= x ) {
         double diffx = x - c[i].x;
         double diffn = c[i+1].x - c[i].x;
```

```
        return c[i].y + ( c[i+1].y - c[i].y ) * diffx / diffn;
      }
    }
    return -1; // Not in Range
}
```

(4) Function which provides a non-blocking delay

```
static bool delay(double delay) {
    static double previous_time;
    static bool done = false;
    static bool first_pass = true;

    if(first_pass){
      previous_time =  wb_robot_get_time();
      first_pass = false;
    }

    double current_time = wb_robot_get_time();

    if(current_time - previous_time > delay) {
      previous_time = current_time;
      done = true;
      first_pass = true;
      printf("\n");
    } else {
      done = false;
    }
    return done;
}
```

(5) Snippet to use the Pen Node (if included in Robot node in the Scene Tree)

```
#include <webots/pen.h>

    WbDeviceTag pen = wb_robot_get_device("pen");
    wb_pen_set_ink_color(pen,0xF01010,0.9);
    wb_pen_write(pen,1);
```

Appendix C Some PROTOs