# Chapter 4
# Navigation

## 4.1 A brief Introduction

We spend a lot of time walking around with a purpose; we have a goal, some place in space we must or desire to arrive at. That's in our mind, and everything else on our journey seems almost irrelevant; we negotiate paths, cross some roads, avoid walls and other folk unless we bump into a friend and stop for a chat; we may interrupt our individual goals, and decide to go for a coffee. To achieve this *navigation*, we use a combination of information; we may know the environment we are navigating through, the familiar corridors and rooms of a building we know. Also, we can adapt to changes in this familiar 'map' we have in our mind, when new furniture is placed along our way, and of course we can avoid other pedestrians.

Robot navigation is not too dissimilar; robots have a current position and a goal position, and our job is to devise algorithms to get them from the one to the other. Like us, they may have a *map* of their environment which they can use to navigate, and they should also be able to avoid obstacles. So, again there are two possible approaches to navigation we have seen before, a ***deliberative*** approach where the robot reasons using a stored map, and a ***reactive*** approach where it uses real-time sensor data to respond to obstacles and openings such as doors. These approaches may be called *planning* and *reacting*.

If the robot has a map of the environment, then the navigation problem is almost totally solved, however this may not be feasible due to constraints on robot memory and processing; this is especially true for the robots we are working with. Remember the microcontroller is dealing with many concurrent tasks; sensing the environment and driving its motors, so there may be little processing power left over to completely analyze a stored map and decide where to go. That’s why we shall focus on *reactive* approaches.

To be able to navigate, our reactive robot needs to know exactly where obstacles are located in relation to the robot’s *pose* (location and heading). The same is true for the detection of openings or gaps between obstacles. We shall start with these situations.

## 4.2 Object Localization

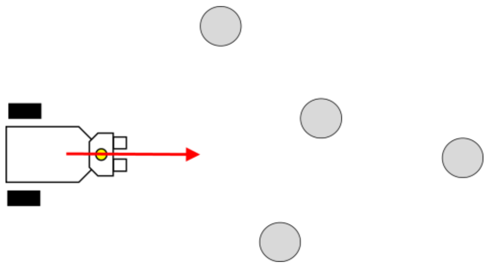


*Figure 4.1 Robot moving to the right in a cluttered field of obstacles*

Consider the robot moving in the cluttered environment shown in Fig.4.1. We can see that the robot is moving towards a definite collision; it needs to *scan* the space it is moving into and detect the obstacle it is most likely to collide with first. So, we must equip our robot with some sort of scanning device, where it will scan the 180-degrees of space in front of it. We can choose various technologies for this scanner, laser, infra-red, but here we shall use our trusty ultrasonic device. The discussion is the same for other scanner types.

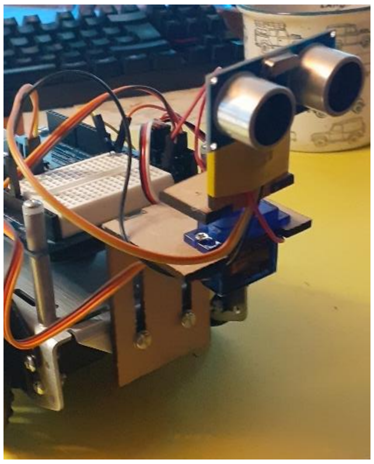


*Figure 4.2 Robot equipped with an ultrasonic detector which can be scanned from right (0 degrees) to left (180 degrees).*

Our robot is shown in Fig.4.2, the us-scanner is mounted on a small servo-motor which can move between 0 and 180 degrees, and we arrange that at 90 degrees it is facing forwards in the direction the robot is facing. Let’s start with the simple problem of localizing a single object in the robots ‘field of view’; so, the robot should find the angle of the object relative to its forward direction, and also the distance to the object. Then it can rotate so it is pointing at the object and then move towards it. This could form a useful application where the robot clears all objects from its arena.

The four stages of this algorithm are shown in Fig.4.3, first the robot (at rest) scans its environment and stores angles and distances to objects it encounters in an array. Then, still at rest it analyses the stored array and looks for the closest object and notes the angle of this object. Stage 3 it rotates so it is facing this object and finally in stage 4 it moves towards this object and pushes it out of the arena. Since there are four clear stages, we use a Finite State Machine architecture.

The array shown below is a part of the entire array of 180 elements; the index into the array is just the current angle. So, in this example the smallest distance is 70mm at the array element 110. This is 110 degrees. In the third diagram we see the robot rotating from its current direction (90 degrees) to the target 110 degrees, in other words it must rotate anticlockwise through an angle 20 degrees.

The code to do this is straightforward; first we must declare an array of integers size 180, since we shall be scanning over a maximum of 180 degrees. The *index* into the array is just the angle.

```
float distArray[180];
```

Then we scan the turret and log the distances

```
for (angle = 20; angle <= 160; angle += 1) {
  servoTurret.write(angle);
  dist = getDistance();
  distArray[angle] = dist;
  delay(60); // Needed by US ping
}
```

Code to analyze the data, to find the closest distance *and to return the corresponding angle is*

```
for (angle = 20; angle <= 160; angle +=1) {
  if (distArray[angle] < minDist &&
distArray[angle] > 0.0) {
    minDist = distArray[angle];
    angleFound = angle;
  }
}
```

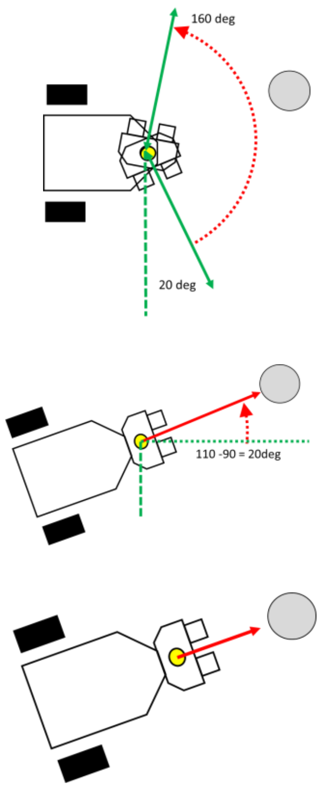


*Figure 4.3 From the top: robot scans the environment and builds a distance array. It finds angle of closest distance and rotates to this angle. Then it moves forwards*

Here is the array of distance values indexed by angle

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 160 | 170 | 170 | 200 | 300 | 350 | 360 | 350 | 360 | 350 | 320 | 100 | 90 | 80 | 70 | 80 | 100 | 410 | 410 | 450 |
| 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 |

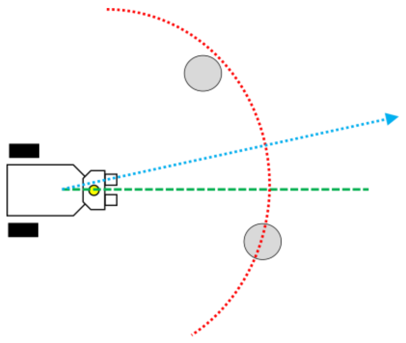


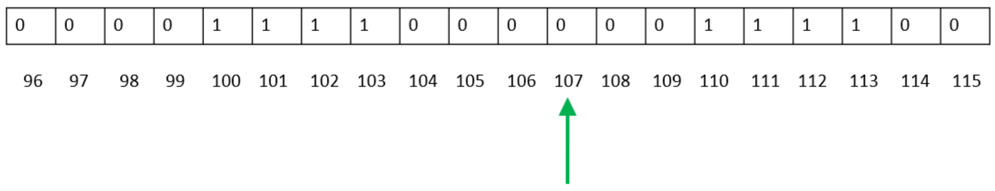
Figure 4.4 Robot detecting gap between objects.

## 4.3 Localization of Openings

Here we are looking at the situation where the robot must navigate to a target location and avoid all obstacles in its path; let's take the case of static objects. This is the situation shown in Fig.4.1. At any time, there will be two objects closest to the robot, so it makes sense to consider how to get the robot to pass through the gap between two objects. This is shown in Fig.4.4. The algorithm to do this is quite straightforward; as in the above example, the arena in front of the robot is scanned. But we do not need to save the distance information (at least present at each angle. We choose a threshold distance (red dotted arc in Fig.4.4) and log in a **labelArray[angle].** If the detected distance is less than the threshold, we put a **1** into the array, else we put a **0** into the array. Here's the code.

```
for (angle = 20; angle <= 160; angle += 1) {
  servoTurret.write(angle);
  dist = getDistance();
  if(dist < distThreshold)
    labelArray[angle] = 1;
  else
    labelArray[angle] = 0;
  delay(60);
}
```

A typical scan could produce an array like the one shown below. In this case the objects are located near 102 degrees and 112 degrees. The centre of the gap between the objects



| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 |

is shown by the green arrow. The index (angle) of this is found by calculating the average of the angles when there is a label of 1. So, we calculate (100 + 101 + 102 + 103 + 110 + 111 + 112 + 113/8 = 107.

Having found this angle, we rotate the robot as in the above example and drive the robot through the gap.

## 4.4 Stopping at the Centre of the Gap

We might think of a situation where we want the robot to find the gap and move towards it but stop at the centre of the gap. This could be developed into an algorithm where the robot navigates through an entire *field* of objects, shown in Fig.4.5. The robot begins at the bottom by scanning and finding the first pair of objects; it moves to the centre of the gap and then stops. Then it performs another scan and finds the centre of the next pair of objects, moves there and repeats.

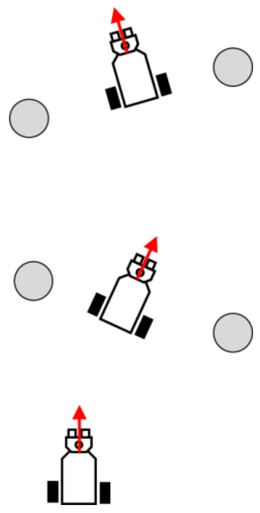
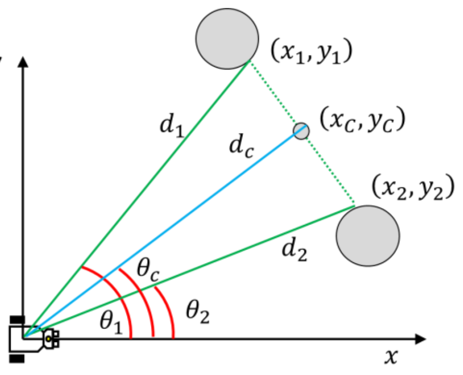


*Figure 4.5 Robot navigating through a field of objects, in stages*

The algorithm to do this is quite straightforward, in fact it is a simple extension (and enhancement) of the gap finding algorithm presented above. All we need to do is, in addition to logging an obstacle detected in the **labelArray[angle]** we need to record the distances in a **distArray[angle].** If we average the distances stored in this array, then this is a good approximation of the distance to the centre of the gap. So, the robot moves at the found angle, but in this case to the distance found. I mentioned that we need to *enhance* the gap-finding algorithm, and to do this we need to consider the geometry of the situation, shown in the sketch below.

The robot is located at the origin $(0,0)$ and we extract from the distance array the distances to the inner edges of the objects, $d_1$ and $d_2$ and their corresponding angles $\theta_1$ and $\theta_2$. Our goal is to calculate the distance to the centre of the gap, $d_C$ and the corresponding angle $\theta_c$. We know how to get this angle, it is the average

$$\theta_C = \frac{1}{2}(\theta_1+\theta_2)$$



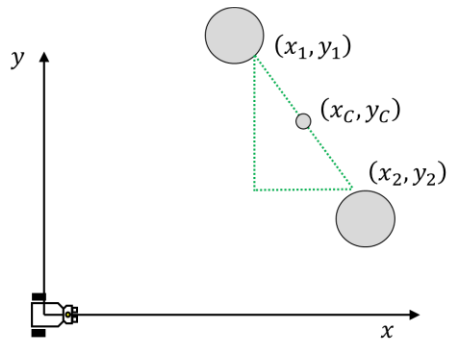
Now we must find expressions for $(x_1, y_1)$ and $(x_2, y_2)$. Using simple trigonometry, we have

$$x_1 = d_1 \cos \theta_1$$
$$y_1 = d_1 \sin \theta_1$$

and

$$x_2 = d_2 \cos \theta_2$$

$y_2 = d_2 \sin \theta_2$Now, we know all distances and angles on the right of all expressions, so we can calculate $(x_1, y_1)$ and $(x_2, y_2)$. All we need to do now is to find the coordinates of the centre of the gap $(x_C, y_C)$. Another diagram may help. Let's think about calculating $x_C$. We start off at $x_1$ and must



add half the width of the green dotted triangle

$$x_C = x_1 + \frac{(x_2 - x_1)}{2}$$

which simplifies to

$$x_C = \frac{(x_1 + x_2)}{2}$$

So, $x_C$ is just the average of the bounding x-values. The same is true for $y_C$, so life has become quite easy here. Once we have $(x_C, y_C)$ then we can calculate the desired distance

$$d_C = \sqrt{x_C^2 + y_C^2}$$

## 4.5 Other Approaches to Navigation

It's worth thinking about other approaches which are possible to implement within the constraints of our small robots.

### The Bug Algorithm

This is the simplest object avoidance algorithm possible. The robot is instructed to move towards a target place, and if it encounters an obstacle on its way, it simply follows the contour of the object by 'hugging its wall' until it can see the target again.