Image Analysis Explanation

Njrb2 | mvs9 | fa296

This document will detail the maths used for the image analysis. This is useful to explain what the code is supposed to be doing when identifying the markers and their orientation. The theory behind identifying the arena boundaries is considerably more simple, and is covered at the end.

# Roomba and pen markers

We use OpenCV’s *findContours* function to find fully-connected, nested lines in the frame. The boxes used in the marker patterns should result in six contours each (three nested boxes, each with an inner and outer contour). Each box will create a hierarchy of at least six contours, so we take all the hierarchies we discover and check them, six at a time, to see if they form a marker pattern.

We identify the corners of each box (the bounds) and the boxes those corners belong to (the outer boxes). We then find the gradient (*m*) and the y-intercept (*c*) for the lines between each point. A special case for these is when the *m = infinity*. This will occur when the two boxes which the line is drawn between are on the exact same x coordinate. The case where *m = 0* will also require special treatment later, when we need to find a perpendicular line for the orientation.

There are six lines in total (top, bottom, left, right and two diagonals). With the line equations, we can check how many times each line crosses a box. Checking whether a line crosses a box is done using the equation *y = mx +* c, where we take the *m* and *c* of the line, and for each box in the pattern we check whether we can get the correct *y* value when we substitute that box’s *x* into the equation. Due to the limitations of floats, we check if the *y* value is correct within a region which extends out from the box’s centre, rather than having to perfectly match the true *y* value. If *m = infinity* then we do some special behaviour, as we cannot perform the maths correctly using *infinity.*

For the Roomba pattern, three of the lines will cross three boxes in total, and three of the lines will cross two boxes in total (see figure x). For the pen pattern, two of the lines will cross three boxes in total, and four of the lines will cross two boxes in total (see figure y). If the number of crossings is different to this, an output of ‘unknown’ is produced.

# Marker orientation

Once a marker has been identified its orientation can be calculated. The orientation is calculated as a value between 0 and 359, where 0 is directly towards the top of the frame. The orientation is effectively calculated in three steps. Firstly, the front edge of the marker is identified – for the Roomba, this is the flat line of three boxes and for the pen this is the flat line of three boxes with a single square on the bottom right.

|  |  |
| --- | --- |
|  |  |

Figure a - The front edge for each marker is identified with a green line.

Once the front edge of the marker has been found, we move onto the second step. A line is drawn between the two outer boxes, and the centre of this line is found. From the centre, we find the perpendicular line. This line runs from the centre of the original line and ends once it hits the edge of the frame.

|  |  |
| --- | --- |
|  |  |

Figure b - The forward facing, perpendicular line for each marker is identified in blue.

We can now move on to the last step, which is applying the sine rule to find the angle. We can construct a right-angled triangle using the perpendicular line from the previous step, and from this we can use two sides and the angle of 90 degrees to find the angle of the marker (see figure c). The sine rule, normally written as when finding an angle, is rewritten in the form and then the *arcsin* function is applied to find the angle.

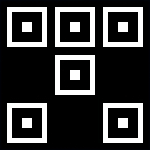


Figure c - The right-angled triangle and the two angles used are shown.

There are four special cases due to lines with *m = Infinity.* If the front edge of the marker is on a line with *m = Infinity* we check whether the back edge is to the left or right of the front edge, and return an angle of 90 degrees of 270 degrees respectively. If the front edge of the marker is on a line with *m = 0* then the perpendicular line will end up having *m = Infinity* and so we check whether the front edge is above or below the back edge, and return an angle of 0 degrees or 180 degrees respectively.

# Boundary Detection

The boundary detection code is the only aspect of the image analysis which was not implemented to be functional in real life, as the decision to focus purely on the simulator was made before the boundary detection was written.

Boundaries are simply pure pink lines (RGB 255, 0, 255) which are detected purely from colour. The shading applied in the simulator means that the actual colour values for pure pink are RGB 155, 0, 155. The shadows in the simulator also reduce the colour values to a minimum of 80, 0, 80. If a line is found within these bounds it is counted as a boundary.

To find a boundary, the pixels coming out from the centre are checked in each cardinal direction (top, left, right, bottom). If a pink pixel is found, that direction is added to an array of boundaries.

Given that the boundaries are likely to be at the edge of the screen, it would usually be more efficient to go from the edge inwards, rather than from the centre out. However, the difference in efficiency is not so large as to be a serious issue, and so for the sake of code simplicity we have left it as going from the centre out.