Sam Davies s1220039 & Michael Inglis s1232123
25 March 2015

IVR Practical 2

Autonomous Navigation

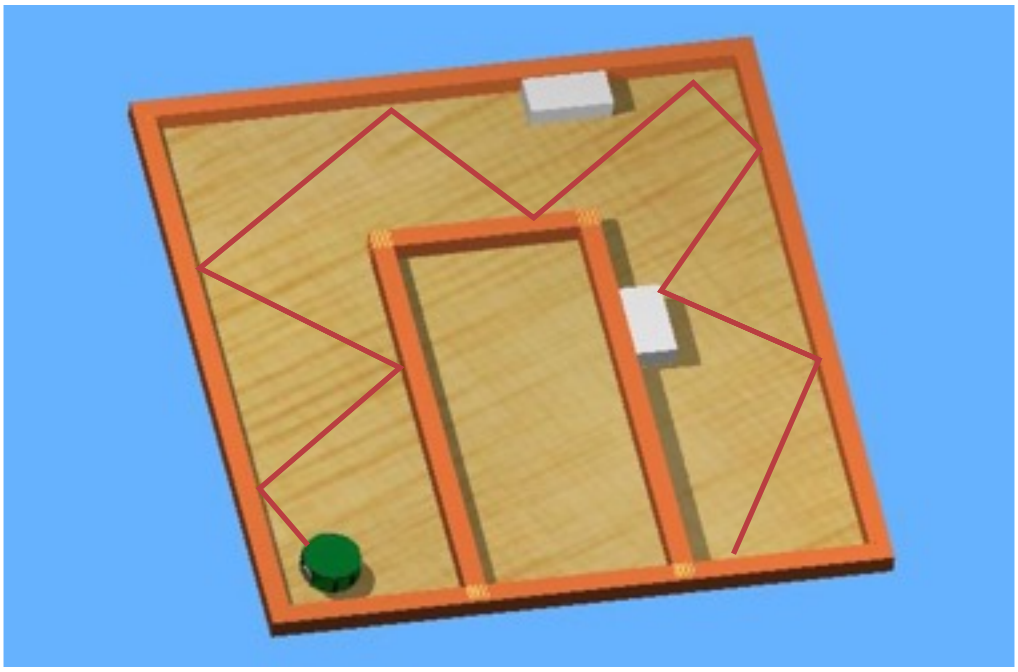
Introduction

In this practical, we were working from Webots using a Khepera robot, programming the robot with Matlab. In an effort to best understand Matlab, as it was our first attempt, Sam thought it would be a great idea to heavily refactor the supplied code into a beautiful object oriented format as well as construction a test suite. Work was split evenly on both the code and report.

Distance Control - Method & Results

The robot was originally set up to turn left and right based on the results from the sensors. If the sum of all the sensors on the left signalled it was close to the wall it would turn right and continue on forward and vice versa for if the sum of all the sensors on the right signalled it was too close to a wall. This was certainly an implementation of the Braitenberg algorithm and the result is as seen below.

Fig 1.1

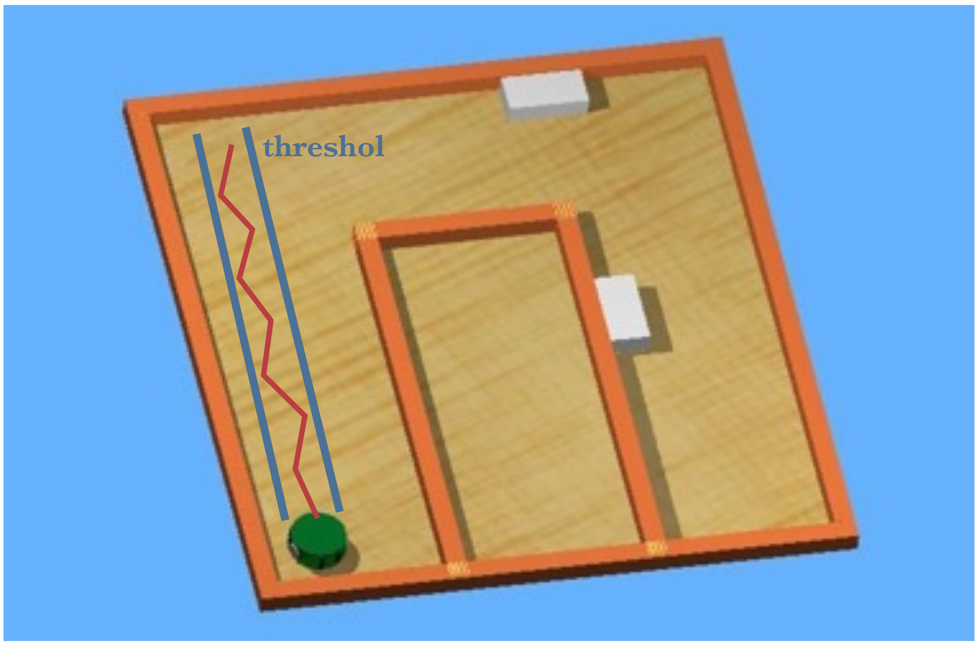


We however decided the task would be easier if we looked at each sensor individually when programming it to stay close to a wall. Our initial testing along the left wall consisted of only using the left sensor, Sensor 1. We basically set a threshold in which the robot attempts to stay in and hug the left wall. If the robot strays outside this threshold and too close to the wall we do 1 of 2 movements:

- turn left, go forwards
- turn right, go forwards

If the robot is within this threshold we move forward. When either of the top sensors come across a wall, the robot turns right until a certain threshold is met. This allows us to deal with 90 degree corners as well as obstacles. This method too allows us to handle the 270 degree corners well too. When sensor one senses we have ran out of wall, it keeps going left until it finds more wall. A visual aid on how our code handles the left wall is is shown below in Fig 1.2.

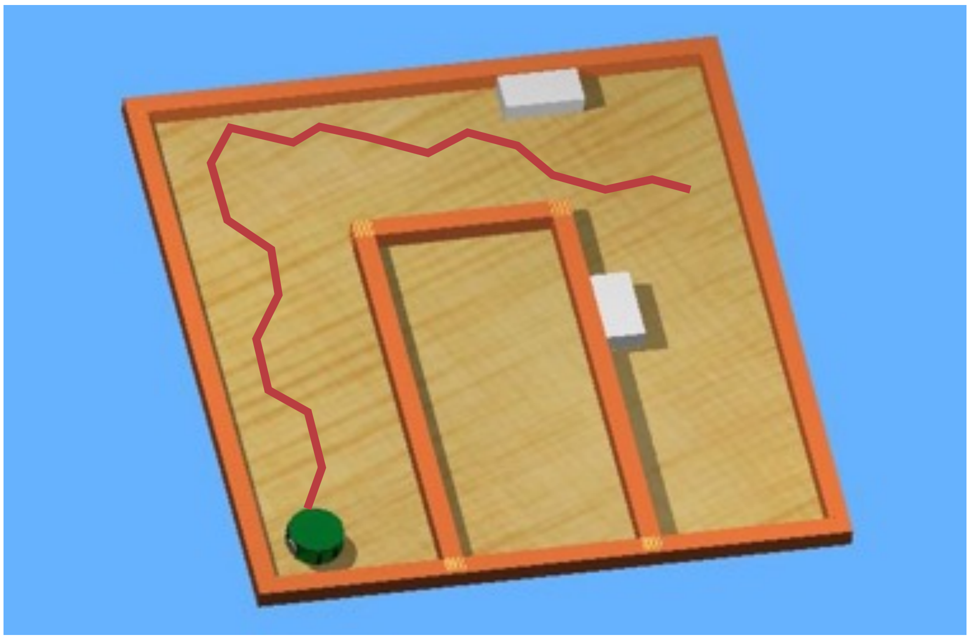
Fig 1.2



The method discussed above was what we achieved our best result with and thus is what is in the code attached at the end of this document.

One of the other methods [fig 1.3] we tried involved a similar threshold method as above, but can only go forwards or a triple of turn-forwards-turn which brings it back parallel to it’s original heading vector. This works fine if the robot starts parallel to the wall, however as seen in the diagram below, if the original vector is no longer the main direction of travel, we start to get erratic results and miss corners.

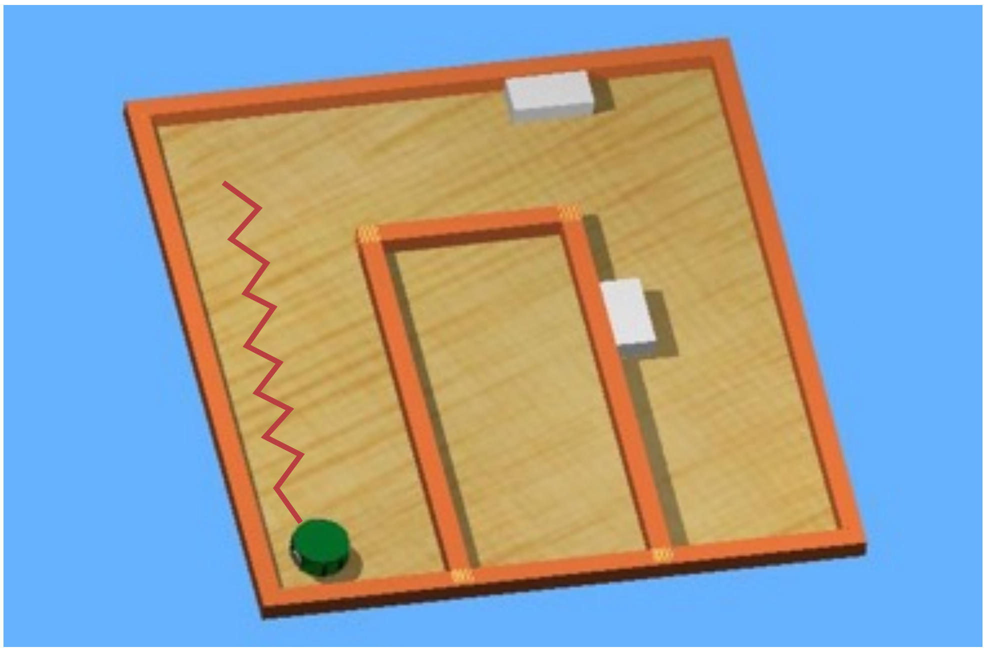
Fig 1.3



Another method we considered is in Fig 1.4. This involves a very similar implementation to our final method, but instead of having the robot go forward as soon as it turns in a direction, this method keeps turning until the sensor is within the threshold. The advantage of this is that it's a simpler algorithm, but this causes the robot to travel a larger distance overall since the 'zig zag' is tighter.

In analysing the dependance of the behaviour on the gain parameters, we concluded that increasing threshold causes the robot to turn less and thus moves faster but doesn't stay as tight to the wall.

Fig 1.4



Odometry - Method & Results

Our odometry was calculated as specified in the slides. Each movement which is carried out has it's own respective direction vector which is added onto the previous position to calculate the current position. We create an odometry object so that we can maintain the state of the position and the direction vectors of the robot. It also holds the functions to then increment the position and rotate the direction vector based on the movement of the robots wheels. Our formula to calculate the angle in which the robot turns was as follows:

Angle robot turns = (distance robot travels/circumference of robot) * 2 pi

where

circumference of robot = pi * diameter of robot

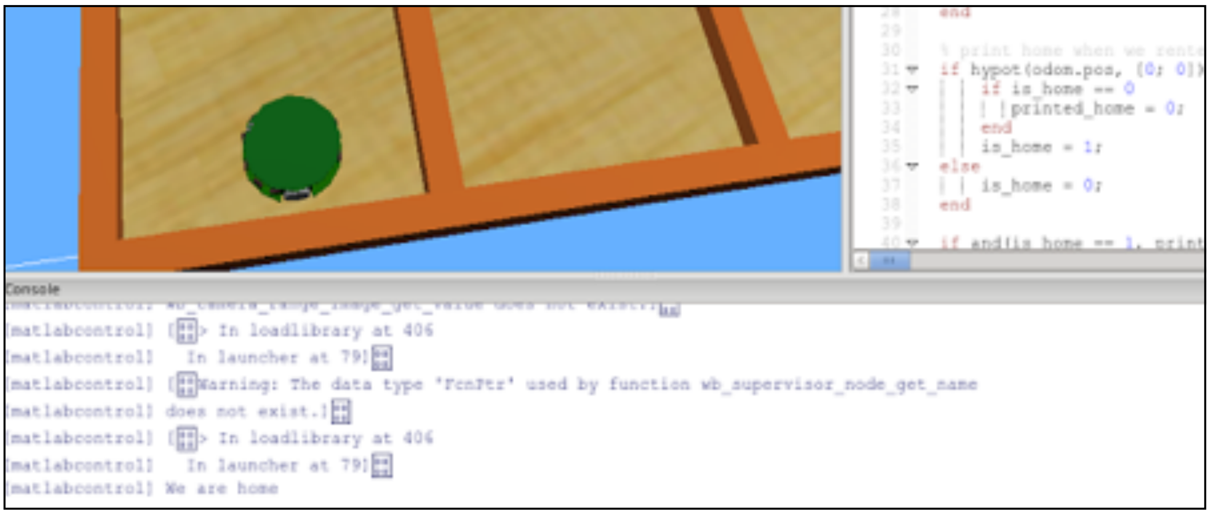
distance robot travels = circumference of wheel * (angle wheel turns/ 2 pi)

where

circumference of wheel = pi * diameter of wheel

angle wheel turns = timestep * angular velocity

We programmed the robot to print out when it reached within a certain threshold of it's original space. Thus showing our robot can calculate where it originally was within a small error. It's position as well as the print statement can be seen below.

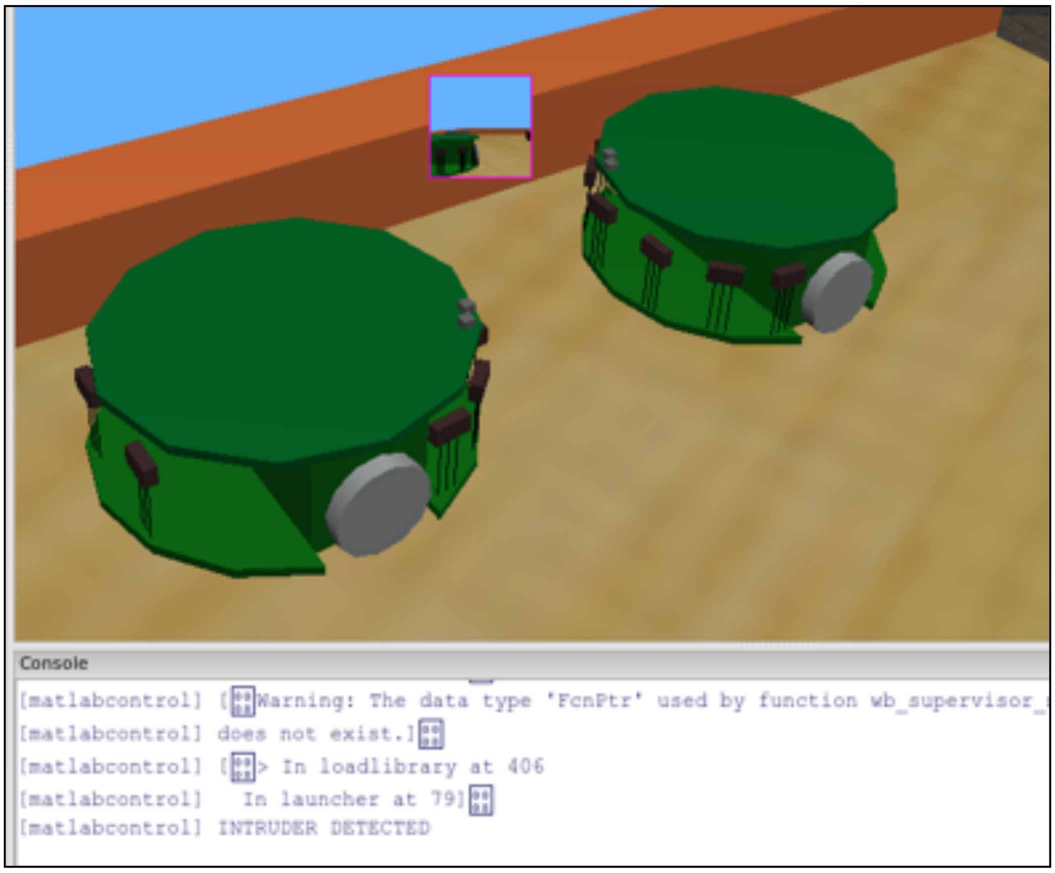


```
end

% print home when we rente
if hypot(odom.pos, [0; 0])
    if is_home == 0
        printed_home = 0;
    end
    is_home = 1;
else
    is_home = 0;
end

if and(is_home == 1, print
```

```
[matlabcontrol] wb_camera_range_image_get_value does not exist.]
[matlabcontrol] [> In loadlibrary at 406
[matlabcontrol]   In launcher at 79]
[matlabcontrol] [Warning: The data type 'FcnPtr' used by function wb_supervisor_node_get_name
[matlabcontrol] does not exist.]
[matlabcontrol] [> In loadlibrary at 406
[matlabcontrol]   In launcher at 79]
[matlabcontrol] We are home
```

Surveillance Robot- Method & Results

Our find intruder method was based on the technique mentioned in number iv. A camera was attached to the top of the robot and using by using the take 64 fps feed, we convert each frame into HSV image and using hue component, we count the number of pixels in green band (between 60 and 180). If the count is greater than 100, we detect an intruder and thus print it to the screen. The result of this can be seen below. Note the virtual camera.



```
[matlabcontrol] [Warning: The data type 'FcnPtr' used by function wb_supervisor_
[matlabcontrol] does not exist.]
[matlabcontrol] [> In loadlibrary at 406
[matlabcontrol]   In launcher at 79]
[matlabcontrol] INTRUDER DETECTED
```

Discussion

All in all we feel our robot adequately tackles the tasks outlined above. The method we went for however disregards most of the sensors on the right side and back of the robot. Our initial plan was to create a state machine with the required movements for each respective result from each one of the sensors, however most of this would be redundant with this assignment.