Introduction to Vision and Robots
Assessed Practical 2

Written by Angus Scott (s1040800) and Eilidh Hendry (s0925284), we would like to share credit 50/50.

Introduction

For this assignment we were given four broad tasks to complete, first was to have our robot follow a straight wall, ensuring it stayed at a reasonable distance from the wall, we accomplished this using Proportional error control, which causes the robot to target an ideal value for some sensors, and will adjust it's motor speed in proportion to the difference between the ideal and actual value of some sensors .

The second was to avoid obstacles using a suitable control scheme, we elected to use the Braitenberg Algorithm as advised in the handout, where the robot moves left if an object is detected to it's right, and vice versa.

The third was to use odometry to implement an algorithm to help our robot to return to it's starting position after it has completed its task, we used the odometry formulae suggested in the final lab, and used those to compute a way of turning towards our start position and then moving there.

Finally, we were to have a robot negotiate around an environment, using some of the code/techniques from the previous tasks.

Methods

We split our code into four distinct sections corresponding to the different tasks in the assignment: part 1 implements wall following; part 3 implements obstacle avoidance, part 4 implements odometry and part b combines the previous parts to have the robot move through a maze.

Part 1 - Wall Following

The robot successfully travels along a straight wall, keeping a reasonably consistent distance between the wall and the robot, with only slight oscillation. This was achieved using proportional error control, where the speed of our motors are adjusted in proportion to the distance of the robot from the wall.

The formula we used for calculating the error, was Err = k*(sensor_value - ideal_sensor_value), where k is our gain parameter used to scale the error, sensor_value which measures the current value output by a particular sensor (or a group of sensors if required), and the ideal_sensor_value which is the sensor value(s) we would like the robot to have.

We then subtract Err from our ideal speed on the right wheel, assuming that the wall is on the left hand side, this allows us to slow or increase the speed of the wheel based on the value of Err, where a positive value or Err implies that the robot is too close to the wall and should therefore have a slower turning right wheel meaning the robot begins to slowly turn right, away from the wall. When we have a negative Err value, this would imply that we are too far from the wall, and we should be moving closer to the wall by increasing the right wheel speed.

We chose to use proportional error control, as it allows for a smoother transition between speeds for different error values, and in turn, a smoother ride for the robot, when compared to an alternative controller such as Bang-bang control. it also allows for smaller oscillations by the robot as the robot is less likely to overshoot it's stable value, at the ideal sensor value.

Part 2 - Analysis of dependence of gain parameters

The parameters we chose were as follows:
ideal_sensor_value = 300, k = 0.001, left wheel = 4, right wheel = 4 - Err

We chose the ideal sensor value because the sensors can take values between 0 and 1000, and only begin to increase from 0 when they are relatively close to the wall, a value of 200 ensured that the robot whilst close, would have a distinct gap between itself and the wall, and would allow space for small amount of oscillation.

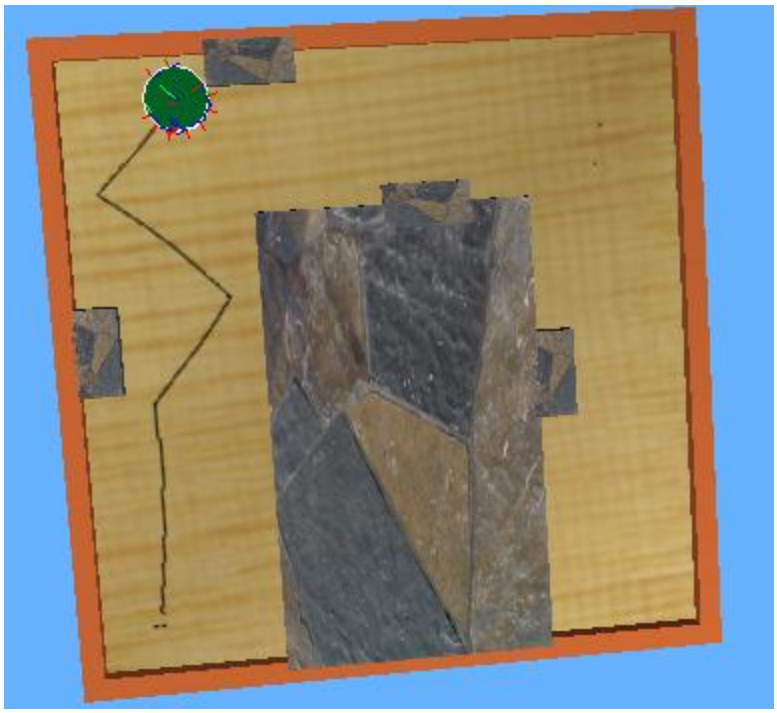
The range of values that s*ensor_value - ideal_sensor_value* are (-300, 700), and since the error has to be scaled with the left wheel speed's value of 4, therefore using the gain parameter k = 0.001, Err has a range of (-0.3, 0.7), these values work well, as it provides a left bias to the robot when no wall is sensed and forces the robot to move away from the wall, when it is to close, with an increasingly strong bias the closer it is.

Part 3 - Obstacle Avoidance

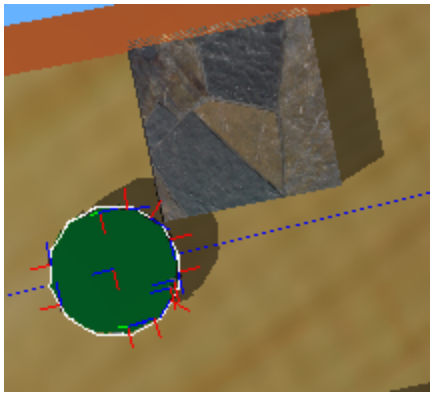
Obstacle avoidance refers to the robot's ability to autonomously respond to objects in it's environment by adjusting it's path. Our robot performs this task by using a simple implementation of the Braitenberg controller. The agent has eight *distance* sensors and two wheels, each driven by their own motor. The sensors are connected to the motors so that a

sensed signal produces a movement of the wheels. The goal of the agent is to avoid obstacles, so when an obstacle to the left is detected, it responds by turning to the right, and vice versa.

In our implementation the robot keeps moving forward, constantly checking the values from the sensors, until it detects an object immediately in front of it, at which point it stops and decides if the object is closer to it's left or it's right and turns the opposite way. In order to decide if an object is immediately in front of it, we sum the front two sensors (sensors 3 and 4) and check if the distance is above a threshold of 1000. To decide if the obstacle is closer to the left or the right we sum all the sensors on the left (sensors 1 to 3) and all the sensors on the right (sensors 4 to 6), then we check if the left sum is greater than the right sum (the object is closer to the left), if so then turn right (the opposite direction), otherwise turn left and move forward. Having successfully avoided the obstacle the robot can continue moving forward until it detects another object in front of it.



We encountered problems that was had not previously anticipated such as the robot getting 'stuck' alongside an obstacle because it's front sensors were detecting the path as clear but there was actually an object immediately to the side of it (as you can see from the image below). So we created a catch for this so that when this situation occurs it knows it is supposed to turn left or right.



Part 4 - Odometry

The Odometry section is split into two parts, an odometry function, that computes the current distance traveled in the x and y directions, and phi, the angle that the robot has turned. And a second that gets the robot to travel some predetermined path, and once completed, computes the angle it should turn to be directly facing home, and then travels in a straight line back to it's starting position.

To compute the current x, y and phi values for the odometry, we use three formulas, similar to those specified in Lecture 12, with the only change being to swap the formulas for x and y around:

x_ = x + 0.5*(left_speed + right_speed)*sin(phi)
y_ = y + 0.5*(left_speed + right_speed)*cos(phi)
phi_ = phi - (left_speed - right_speed)/ (2*R)

where left_speed and right_speed are the values of the wheel speeds in Rad/s and R is an optimised parameter which would normally denote the radius of the robot, however as our wheel speed is in Rad/s and not in cm/s, we has to use a different value than the one suggested in the lab, and a value of R = 53 was found to produce accurate results.

To perform homing, we had to make the assumption that there would be no objects in the way between the point where we completed our tour, and our origin. This was done as the amount of work required to have a robot detect the object, move around it and then resume homing would have been too difficult and time consuming for the project to have been completed in a reasonable time.

In order to complete homing, we required the robot to rotate from it's current bearing to the bearing facing it's origin. The approach we took, was to calculate tan^-1(O/A), where O would be the absolute value of x and A the absolute value of y, if x and y were greater than 0, or x and y

were less than 0. Otherwise O would be the absolute value of y and A would be the absolute value of x.

Then to compute the angle phi the robot should be facing, we computed the following:

if x < 0 and y < 0, then orientationHome = tan^-1(O/A)
if x < 0 and y < 0, then orientationHome = tan^-1(O/A) + pi/2
if x > 0 and y > 0, then orientationHome = tan^-1(O/A) + pi
if x > 0 and y < 0, then orientationHome = tan^-1(O/A) + 3pi/2

We then get the robot to turn to that angle, and then move in a straight line back to it's origin, checking for when x and y are very close to 0.

Part 5 - Analysis of dependency of odometry error

The x, and y values are reasonably accurate for a robot that moves in a straight line and back, any error introduced in the system is introduced by the calculation of the angle using the parameter R. When the robot moves in a straight line and back, the angle is not changed, and therefore is reasonably accurate, however as soon as the robot turns and is dependent on R, error is introduced, and whilst R is reasonably well estimated, it can be difficult to get absolutely correct. However when performing the experiment below, it can be seen that the robot odometry is still reasonably accurate, even with large distances. But of course, even with small errors, as you increase distance traveled, the small mistakes become larger due to compound errors.

One way to gauge the error, is by having a robot complete two different paths between two points. For example if you have the robot perform a random path, and once it has completed that path, take the shortest route home, you can get a sense of the error by viewing the difference between the starting point on the random path, and it's end point on the shortest path as two different paths that have the same start and end points should have the same x and y values at beginning and end. The results using this test are detailed in the Results section.

Part B - Surveillance Robot

The task in Part B was for the robot to follow a sentry path around a C-shaped hallway in order to check for any robot intruders. The path is obstructed by obstacles which are supposed to be circumvented by the robot. Additionally, the robot should finish at the starting position after completing one lap around the environment. This task was completed by combining the code from the first sections.

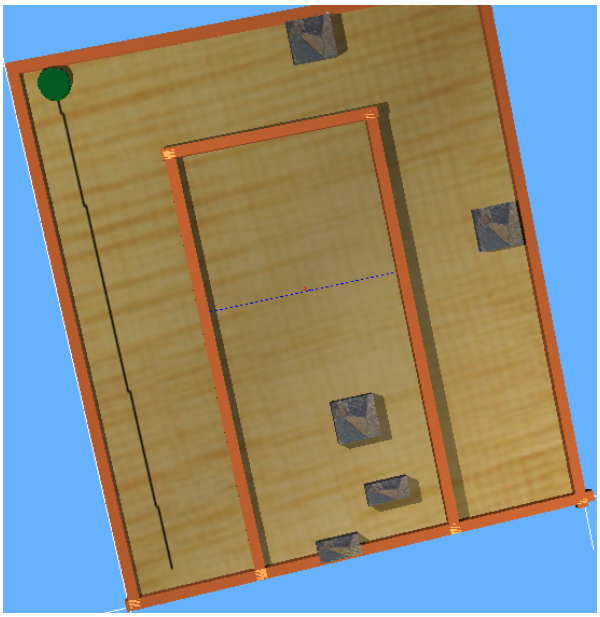
Combing the sections involved using a subsumption architecture which means the robots tasks are separated into ‘layers’ and prioritised using nested if statements. These layers roughly correspond to tasks 1, 3 and 5. The lowest layer involves the robot simply following the wall. Then on top of that we made it constantly check for obstacles and avoid them when they are encountered. Finally, we use the odometry to make the agent stop when when it reaches its starting point again.

Without making any changes to parts 1-3 when they were combined the robot could already follow the wall and navigate around corners and around objects. However, in order to get it to perform optimally we had to experiment with different values for the error distance and K (further discussion in results). These values were less important when the robot was simply following a straight wall, but when the robot was required to navigate through smaller spaces and back around obstacles it was necessary to find the parameters that worked best.

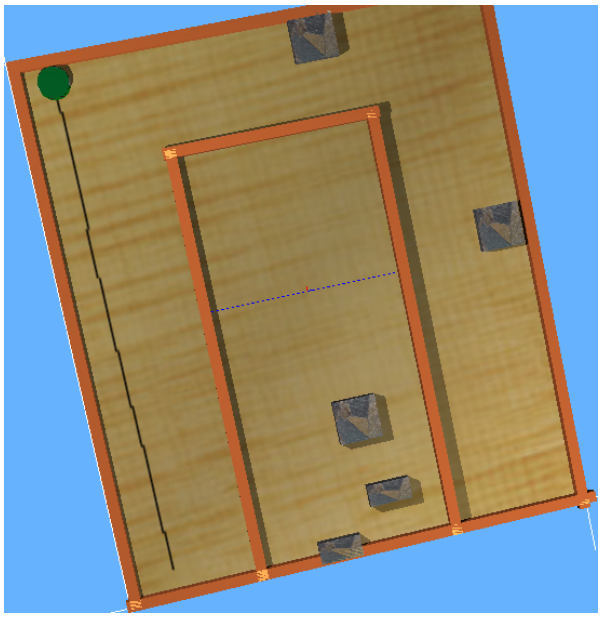
Results

Varying Value of K

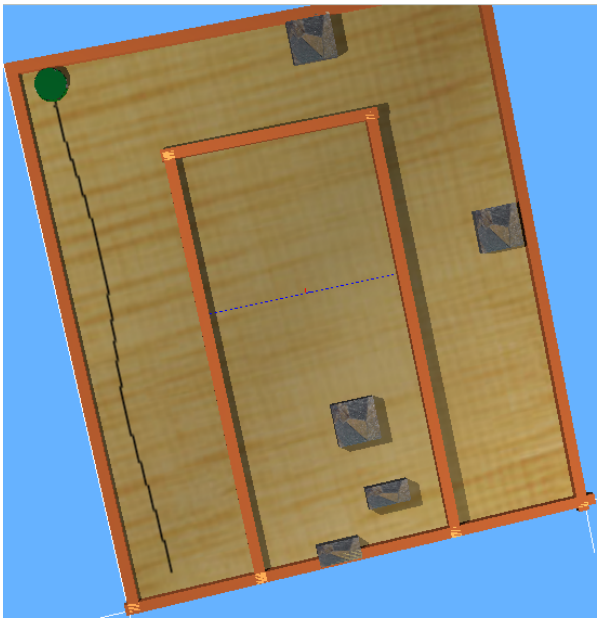
In order to evaluate the effect of varying the value for the parameter K we kept the value for the ideal sensor value constant at 300. We attached a pen object to the Kheprera so it could draw the path it had taken and clearly see the effect of varying the parameters. The results of varying the K are shown in the images below.



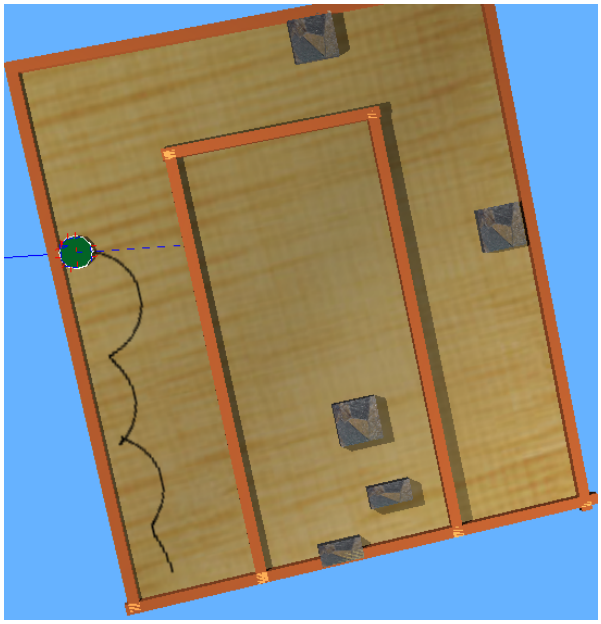
K = 0.00001



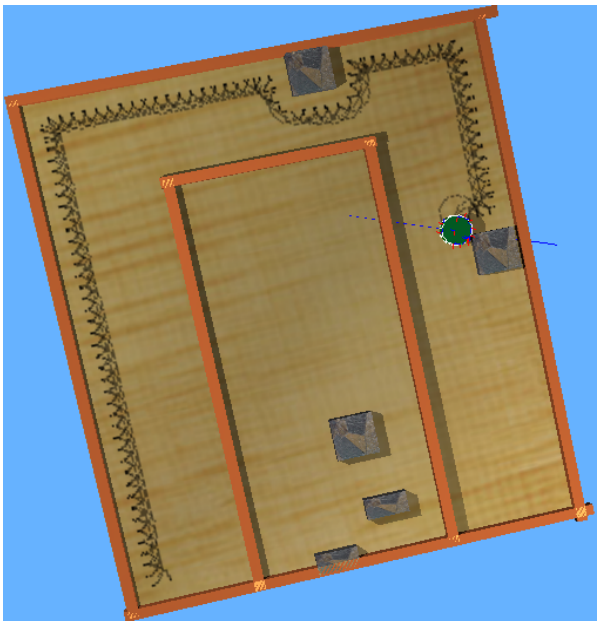
K = 0.0001



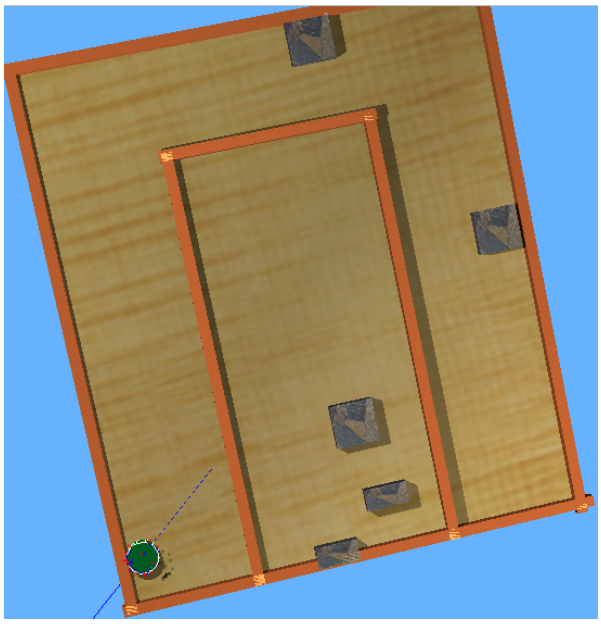
K = 0.001



K = 0.01



K = 0.1

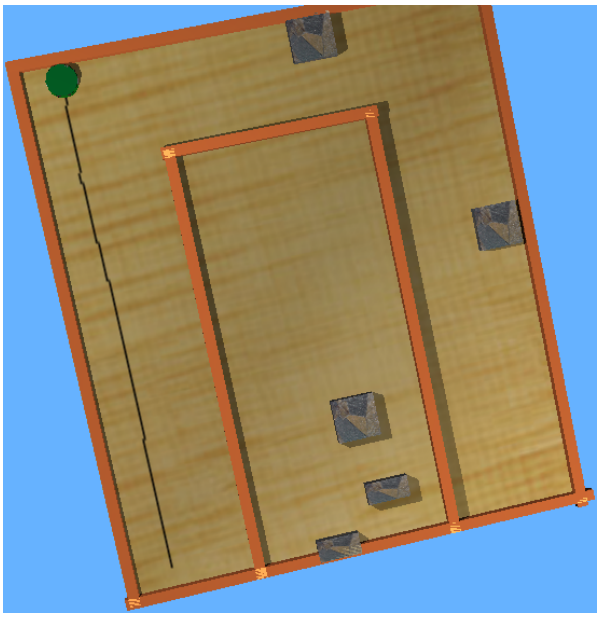


K = 1

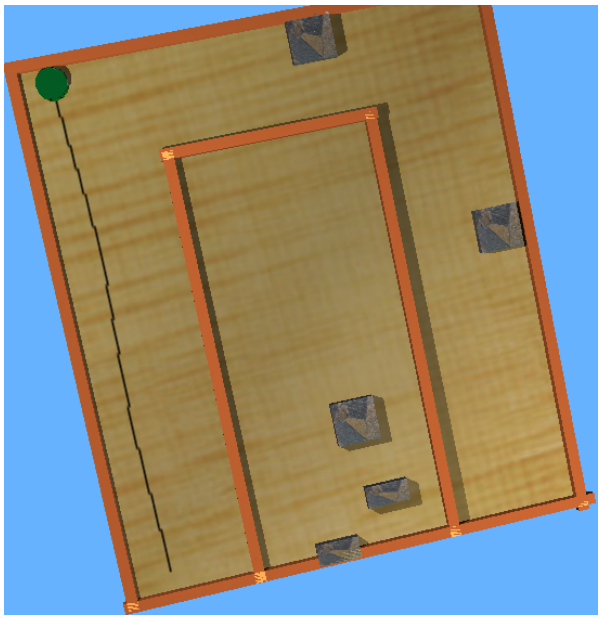
We can see from the results that with values of 0.00001 to 0.001 (the first 3 images), the robot can successfully follow the wall. However, with values of 0.01 successfully follow the wall and instead steers into the wall and gets stuck. Surprisingly, a value of 0.1 works well in that the robot can follow the wall and even navigate around objects and walls. However, it does this in a very slow ineffective way. With a value of 1 the robot immediate swerves straight into the wall.

Varying the ideal sensor value:

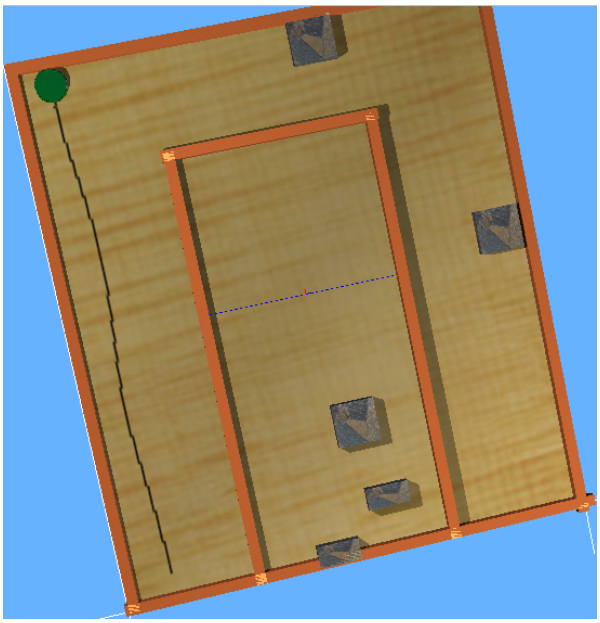
The ideal sensor value is the value used in the proportional error control as described above. In order to see the effect of varying the ideal sensor value we kept all the other parameters constant including the value for K which we kept at 0.001. In this experiment we also used the pen attachment to draw the path taken by the khepera.



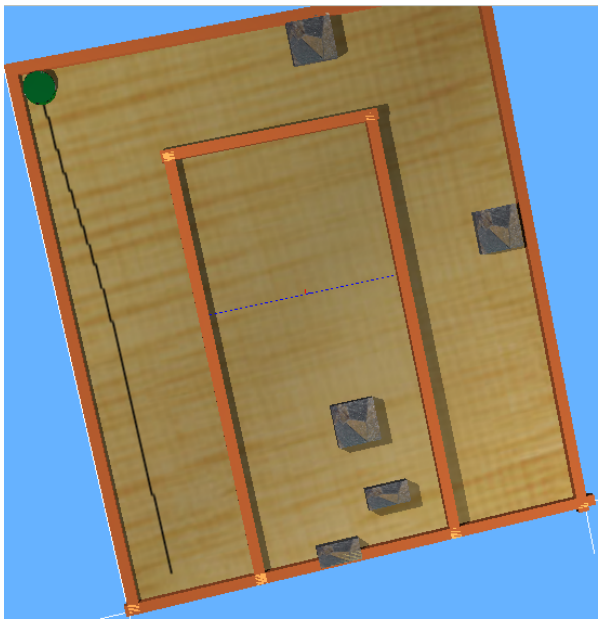
sensor value =100



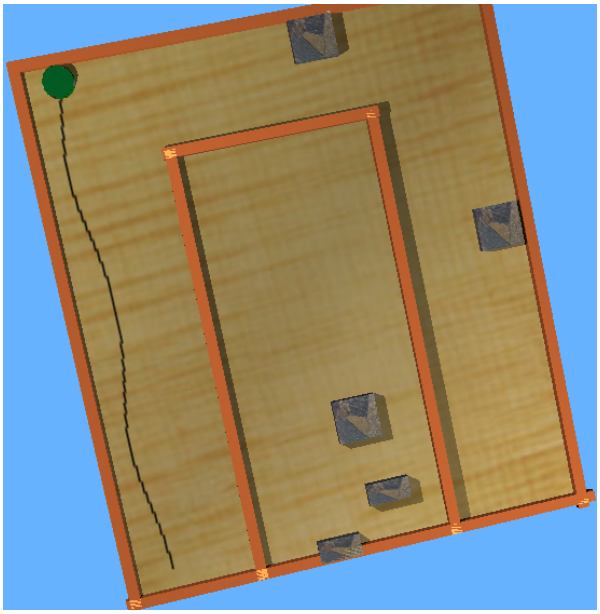
sensor value = 200



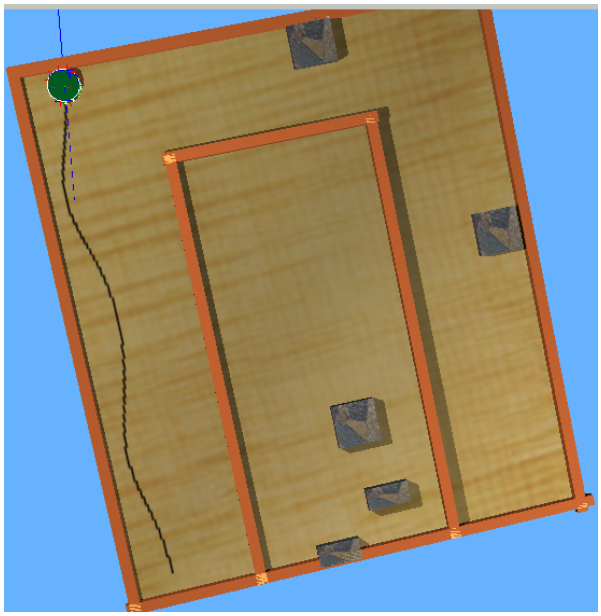
sensor value = 300



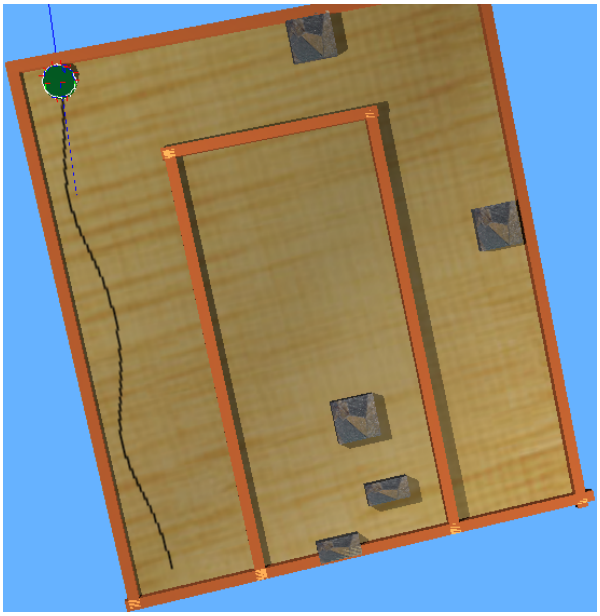
sensor value = 400



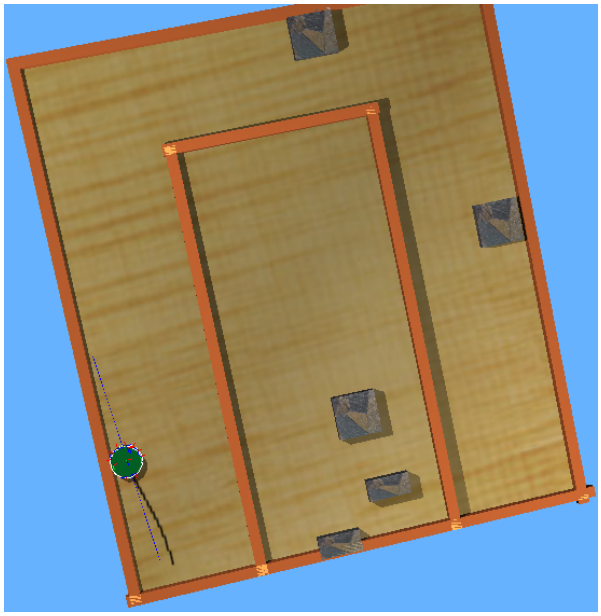
sensor value = 500



sensor value = 600



sensor value = 700

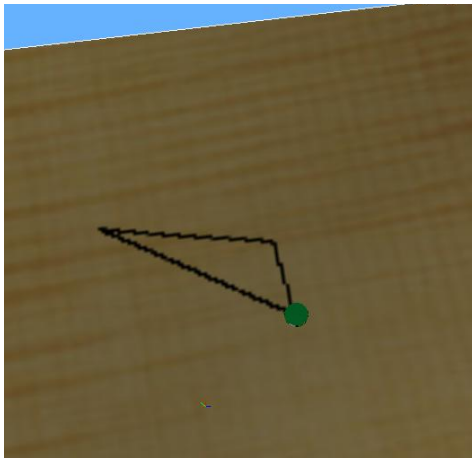


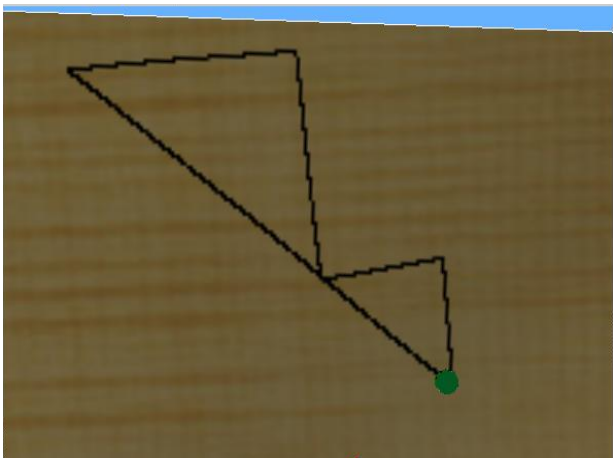
sensor value = 800

We can see from the results that varying the ideal sensor value affects the oscillation of the path taken by the robot. A higher sensor value means a greater degree of oscillation. The robot manages to successfully follow the wall except for when the ideal sensor value is 800, when it steers left so much that it steers into the wall.

Odometry Experiment

Using the technique outlined in Part 5 of our Methods section we can test the accuracy of the Odometry class, and it's ability to return back to the point it started at, some sample tests are below, and the long straight paths are where the robot has turned and went straight back to it's starting position, as we can see the results indicate that even with the robot traveling relatively large distances, there is very small amounts of error.





Unfortunately it is difficult to test the performance of the Odometry class in Part B as the robot doesn't always complete the course and won't end up near where it started, however as the tests above illustrate, the odometry does work, and works in a variety of different situations.

Discussion

From our experiments we can conclude that our robot performs well on the majority of the tasks outlined in the assignment brief. It can demonstrate stable wall following behaviour. Additionally, we can see the control scheme for obstacle avoidance works well allowing the robot to navigate around obstacles and escape from a position facing a corner. It can also keep track of it's position relative to a defined starting position by odometry and use the odometry to guide itself back to it's starting position.

Although our robot performs all the tasks required for Part A, it is not always successful at completing the task outlined in Part B.

It is able to follow the walls, navigate around objects and keep track of it's starting position. However, it does not always successfully complete a full lap of the c shaped map. Sometimes, this happens because the robot ends up in a situation where it is facing a corner but at the wrong angle, and because the object on the left is closer to the object on the right it turns back on itself meaning it does not complete a full circuit of the environment.

Another problem we encountered was that occasionally the robot would get stuck in a position where all it's sensors were past an obstacle but the widest part of it's body was not. In this

situation the robot would keep trying to move forward but would be unable to. A possible fix for this situation would be to store the position of the robot and if it has not moved for the previous 5 iterations, then reverse and turn.

An additionally improvement we could have made to the performance of our robot is we could have used a more advanced robot control method such as PID control. Instead of just using Proportional error control, we could have included proportional Integral control and proportional Derivative control.

However, despite having a relatively simple robot control method we found that our agent could perform most of the tasks that we set it. A more complex method may not give a significant increase in performance.

A final suggested improvement would be to include a memory for the robot, so it can store a map of it’s environment and successfully search for intruders by following the map.