# CS26410 - Assignment 3
# Robotic Hide and Seek

Samuel Jackson
slj11@aber.ac.uk

April 18, 2013

# Contents

# 1 Introduction

Assignment three for CS26410 required us to develop a collection of robotic controllers for use with the Pioneer robots in the ISL which could be used to play a game of hide and seek. This document details the solution that I have provided along with a discussion of its performance using trial runs on real robots and an evaluation of the work done.

# 2 Discussion of the Algorithms Used

In the development of this assignment I have used several different techniques and algorithms to solve the problems outlined in the brief. For each of the major problems to be solved in the brief I have supplied a description of my approach to finding a solution.

## 2.1 Mapping Techniques

The first and most important component of the solution was to be able to create a map of the environment. The easiest approach to create a simple map of the environment was to create an occupancy gird of the world split into 60x60cm cells in which the robot could fit. As the robot moves through the world, it checks the readings from its sonar array and updates the likelihood that a cell is a wall using the range measured from the sensors. When the robot receives a reading that falls outside the area of the existing grid, my program will dynamically resize the grid to add more cells in the direction of the new reading.

Listing 1: Modified depth-first search algorithm using both in mapping and during localization.

```
1  //while there are still unexplored cells.
2  //or we haven't yet managed to localize
3  while((!frontier.empty() && !localize) || (!frontier.empty() && !localized)) {
4      current->SetValue(0);
5
6      //find the 4 neighbours
7      threshold = grid.CalculateThreshold()/2;
8
9      vector<Cell*> neighbours = GetNeighbours(current);
10
11     //select new direction if required.
12     for (int i=0; i <= 4; i++) {
13         direction = (direction + i) % 4;
14         if(neighbours[direction]->GetValue() <= threshold && !neighbours[direction]->IsVisited()) {
15             //valid cell to move too.
16             backtracking = false;
17             nextCell = neighbours[direction];
18             nextCell->SetValue(0);
19             nextCell->SetDiscovered(true);
20             break;
21         } else if(i >= 4) {
22             //find cell on frontier
23             backtracking = true;
24         }
25     }
26
27     //add neighbours to frontier for later examination
28     for(vector<Cell*>::iterator it = neighbours.begin(); it != neighbours.end(); ++it) {
29         Cell *neighbour = *it;
30         if(neighbour->GetValue() <= threshold) {
31             // cout << *neighbour;
32             if(!neighbour->IsDiscovered() && !neighbour->IsVisited()) {
33                 //add new cell to frontier and mark discovered
34                 neighbour->SetValue(0);
35                 neighbour->SetDiscovered(true);
36                 frontier.push_back(*it);
```

```
            }
        }
    }

    cout << "MOVING FROM/TOO:" << endl;
    cout << *current << endl;
    cout << *nextCell << endl;

    //if we are heading backwards
    if(backtracking) {
        //get next non visited cell on frontier.
        while(!frontier.empty()) {
            nextCell = frontier.back();
            frontier.pop_back();

            if(!nextCell->IsVisited() && nextCell->GetValue() <= threshold) {
                //find path from this square to us.
                MoveToCell(current, nextCell);
                current = nextCell;
                break;
            }
        }
    } else {
        //just move to the next cell.
        MoveToNextCell(*current, *nextCell);
        nextCell->SetVisited(true);
        current = nextCell;
    }

    //if we're not localized, attempt to
    if(!localized) {
        myLocation = Localize();
    }

}
```

In the previous assignment this mapping was done using a purely random walk. While this approach worked well, it had the major drawback of not having a way to finish mapping unless a human actively told it to stop. In this assignment I decided to take a more structured approach to mapping the environment by utilising depth-first and A* search techniques shown in the listings above and below.

Using the new mapping algorithm the robot moves 60cm at a time from one cell to the next while still reading input for its sensors and updating the occupancy grid. When the robot moves into a new cell it checks each of its neighbours to see if they have been discovered or visited. If neither of these conditions are true the robot will add the cell to its "frontier" to come back and look at later. The robot then attempts to keep moving forwards unless there is an obstacle in the way. If there is an obstacle in the way then it will attempt to move into the cells on the left or right of it. If the robot hits a dead end it will take the next cell off the frontier and perform an A* search to find the quickest path from the current cell back to the next unexplored cell. The mapping algorithm will continue in this way until there are no more unexplored cells and hence the map is complete.

Listing 2: A* search algorithm used in path finding and mapping.

```
vector<Cell*> Mapper::FindPath(Cell* start, Cell* goal) {
    //define map of f scores, g scores
    map<Cell*, int> f_score;
    map<Cell*, int> g_score;

    //close and open sets
    vector<Cell*> closed_set;
    vector<Cell*> in_queue;

    //map for paths reoonstruction
    map<Cell*, Cell*> came_from;

    //finished path to follow
    vector<Cell*> path;

```

```
16      //Priority queue of cells in frontier
17      //ordered shortest distance to goal first
18      priority_queue<Cell*, vector<Cell*>, ComparePoints> frontier(ComparePoints(goal, f_score));
19
20      //add where we are to frontier
21      frontier.push(start);
22
23      //Init start cell scores
24      g_score[start] = 0;
25      f_score[start] = g_score[start] + ComparePoints::Distance(start, goal);
26
27      Cell* current;
28
29      //while there are still potential pathways
30      while (!frontier.empty()) {
31          //get next best looking cell
32          current = frontier.top();
33
34          //if its the goal, we're done!
35          //Reconstruct the path and return it
36          if(*current == *goal) {
37              return ReconstructPath(came_from, goal);
38          }
39
40          //else add it to closed set and examine it
41          closed_set.push_back(current);
42          frontier.pop();
43
44          //get the neighbours of the current cell
45          vector<Cell*> neighbours = GetNeighbours(current);
46          for (vector<Cell*>::iterator it = neighbours.begin();
47              it != neighbours.end(); ++it) {
48              Cell* neighbour = *it;
49
50              //check if the neighbour is a valid cell
51              if(neighbour->GetValue() <= threshold) {
52                  int tentative_g_score = g_score[current] + 1;
53
54                  //if we've already examined it and it's score isn't any better just forget
55                  //and contiune with next neighbour
56                  if(vec_contains(closed_set, neighbour)) {
57                      if(tentative_g_score >= g_score[neighbour]) {
58                          continue;
59                      }
60                  }
61
62                  //if its not already in the queue to be looked at
63                  //or has a shorter path score
64                  if(!vec_contains(in_queue, neighbour)
65                      || tentative_g_score < g_score[(neighbour)]) {
66                      came_from[neighbour] = current;
67
68                      //compute the new scores for this neighbour
69                      g_score[neighbour] = tentative_g_score;
70                      f_score[neighbour] = g_score[neighbour] + ComparePoints::Distance(neighbour, goal);
71
72                      //if it's not already queued, queue it!
73                      if(!vec_contains(in_queue, neighbour)) {
74                          frontier.push(neighbour);
75                          in_queue.push_back(neighbour);
76                      }
77                  }
78              }
79          }
80      }
81
82      //return the empty path if no path found
83      return path;
84  }
85
86  vector<Cell*> Mapper::ReconstructPath(map<Cell*, Cell*> came_from,
```

```
87      Cell* current_node) {
88
89      vector<Cell*> vec;
90      //if current node in the map
91      if(came_from.find(current_node) != came_from.end()) {
92          //recursively build the map
93          vec = ReconstructPath(came_from, came_from[current_node]);
94          //add this node to the end of the path
95          vec.push_back(current_node);
96          return vec;
97      } else {
98          //just add it to the end of the path
99          vec.push_back(current_node);
100         return vec;
101     }
102 }
```

This technique proved to be fairly suitable to the problem as it ensured that, presuming the world is closed, the robot would eventually map all of it. To ensure that the same parts of the map would not be remapped I turned off the map updating function when the robot was backtracking. While I realise that the use of the A* function is not strictly necessary as backtracking could be implemented with a simple depth-first search, it speeds up mapping process considerably by not having to move back through every single previous cell, but instead takes the shortest route to the next unvisited cell.

## 2.2 Localization

After a map of the environment has been created it can be used to aid the robot when hiding or seeking. Note that my map making application saves the generated map as a CSV file which can be loaded into the hiding and seeking programs for reference later.

When I start the hider application, it loads the saved map from file and converts it to a map consisting of 0, 1 or -1; where 0 is a floor, 1 is a wall and -1 is "undefined" space which we can ignore. To convert the raw map (containing the original floating point values) to a more structured map I use a threshold value to ensure that cells which might have been incorrectly marked as being slightly occupied are not treated as walls. In this project I am using Otsu's method of thresholding to find the point that minimizes intra-class variance between what can be labelled as a wall and a floor.

Listing 3: Function for the localization of a robot.

```
1  Point Mapper::Localize() {
2      Point me;
3      me.SetX(-1);
4      me.SetY(-1);
5
6      //only attempt to localize when the grid expands some more
7      //(or at the start)
8      if(grid_height != grid.GetGridHeight()
9          || grid_width != grid.GetGridWidth()) {
10
11         int matches = 0;
12
13         //loop over exisitng map
14         for(int i =0; i < map_height - grid.GetGridHeight(); i++) {
15             for (int j =0; j < map_width - grid.GetGridWidth(); j++) {
16
17                 int count = 0;
18                 //at each point, check if our grid matches
19                 for(int k = 0; k < grid.GetGridHeight(); k++) {
20                     for (int l = 0; l < grid.GetGridWidth(); l++) {
21                         double mval = mapData[i+k][j+l];
22                         double gval = grid.GetCell(l,k)->GetValue();
23
24                         //check if cells match
25                         if((gval == 0 && mval == 0) || (gval > 0 && mval > 0)) {
```

```
                        count++;

                    //count it as matched if it's unvisited
                    } else if(gval < 0) {
                        count++;
                    }
                }
            }

            //if we match every square
            if(count == (grid.GetGridWidth()*grid.GetGridHeight())) {
                Cell* loc = grid.GetCurrentCell();
                me.SetX(loc->GetX()+j);
                me.SetY(loc->GetY()+i);
                matches++;
            }
        }
    }

    grid_width = grid.GetGridWidth();
    grid_height = grid.GetGridHeight();

    cout << matches << endl;

    //if we have exactly one match we're localized
    if(matches == 1) {
        localized = true;
    }
}

cout << *grid.GetCurrentCell();
cout << "Point x: " << me.GetX() << " y: " << me.GetY() << endl;

//return the point we think we're at
return me;
}
```

In order to localize the robot we can start by mapping our environment to build up a picture of what is currently around the robot. I can then use this information and compare each cell in the new map with the existing map to see if it uniquely matches up. If the there is a single unique match then we have localized the robot inside the map. If there are no matches or multiple matches, the program will continue mapping until the grid expands and then attempt to localize again, or until the entire world has been mapped (if we've mapped the whole world again then we clearly must be localized in the environment).

This method works on the basis that at some point in the map there will emerge a definitive landmark that can only match as single arrangement of cells in the pre-existing map. However, there are several problems with the approach that make it less suitable when deployed in the real world. One is the fact that the time complexity of comparing the whole pre-existing map to the newly generated map multiple times is hardly efficient. Secondly, this approach will not work if the robot is not placed such that the newly generated grid matches up with the existing map. For example, imagine a corridor three cells wide. The robot could be put in such a position that it thinks the corridor is only two cells wide. This localization method would fail in this scenario because this section of the map would never sync up with the existing map and the localization function would never return a match.

## 2.3 Finding a Hiding Spot

Once a reference map has been generated and the robot can localize within it, the robot can now find a good spot to "hide". In my solution, a hiding spot is found by evaluating all floor cells on the grid and checking how far away each of the neighbouring walls is. The further away the wall is the more visible the square is and hence higher its score will be. After evaluating all potential positions the program simply picks the cell with the lowest score and uses A* to find the shortest path from the current position to the hiding spot.

Listing 4: Function for finding a hiding spot for the robot.

```
Point MapLoader::FindHidingSpots(const std::vector<std::vector<int> >& mapData) {
    int minScore = 16;
    Point p;

    //for every cell in the grid (ignoring the buffering around the outside of the map)
    for (int i = 2; i< mapData.size()-2; i++) {
        for (int j = 2; j< mapData[i].size()-2; j++) {

            //certified big number
            //guaranteed to be large
            int score = 999999;

            //if valid square
            if(mapData[i][j] == 0 && mapData[i-1][j] >= 0 && mapData[i][j-1] >= 0&& mapData[i+1][j] >= 0 && mapData[i][j+1] >= 0) {

                //get all the neigbours 4 away from each cell in +/-x and +/-y

                for(int k = j; k <= j+4; k++) {
                    if(j+4 >= mapData[i].size()) {
                        break;
                    } else if (mapData[i][k] == 0) {
                        score++;
                    } else {
                        break;
                    }
                }

                for(int k = i; k <= i+4; k++) {
                    if(i+4 >= mapData.size()) {
                        break;
                    } else if (mapData[k][j] == 0) {
                        score++;
                    } else {
                        break;
                    }
                }

                for(int k = j; k >= j-4; k--) {
                    if(j-4 < 0) {
                        break;
                    } else if (mapData[i][k] == 0) {
                        score++;
                    } else {
                        break;
                    }
                }

                for(int k = i; k >= i-4; k--) {
                    if(i-4 < 0) {
                        break;
                    } else if (mapData[k][j] == 0) {
                        score++;
                    } else {
                        break;
                    }
                }

                //if this score lower, set it as the new point to hide at
                if(score < minScore) {
                    minScore = score;
                    p.SetX(j);
                    p.SetY(i);
                }
            }
        }
    }

    //return the suggested hiding spot
    return p;
}
```

This solution seems like fairly reasonable considering the technology available on the Pioneers. There is only very limited information that can be gleaned the Pioneer robots with which to base a solution on and this method is always guaranteed to return a decent solution regardless of the size or shape of the environment providing the map and localization is correct. Obviously, this technique assumes that there is no specified "seeker" start point.

### 2.4 Finding Map Differences

The final major problem to be solved was finding a hidden robot within a new map when compared with a reference map. To solve this problem my robot will simply remap the environment and load the reference map, apply their respective thresholds and then compare each point in each map to check if it has changed. The program then outputs a list of points that differ between the two maps. This solution works well provided the threshold values are reasonably accurate the the two maps are also fairly accurate, but could suffer from the same issues as the localization function where the new grid doesn't exactly match the old one.

Listing 5: Function for finding differences in the map.

```
1  std::vector<Point> MapLoader::FindNewCells(const std::vector<std::vector<int> >& map1,
2      const std::vector<std::vector<int> >& map2) {
3
4      //vector of points that have changed
5      std::vector<Point> diffPoints;
6
7      //loop over old map and compare each point with the new map
8      for(int i=0; i< map1.size(); i++) {
9          for (int j=0; j < map1[i].size(); j++) {
10             if(map1[i][j] != map2[i][j]) {
11                 Point p(j,i);
12                 diffPoints.push_back(p);
13             }
14         }
15     }
16
17     return diffPoints;
18 }
```

## 3 Trail Runs

### 3.1 Making a Map

This section documents a number of trials runs of my application carried out on the Pioneer robots in the ISL. For the first trail run, I ran the map maker program to get the Pioneer robot to generate a map of the environment. This run was carried out in a simple environment shown in the following image.

Figure 1: Pioneer robot at start point in a simple environment.

The robot then proceeded to map the environment by moving approximately 60cm at a time. The underlying movement system does not require the robot to move exactly 60cm forwards and if it does not move exactly this distance then it will add the difference onto the next movement to help correct for errors over time.

Figure 2: Pioneer robot after moving forward in the environment.

Likewise, when the robot turns it only attempts to turn within 5 degrees either side of the target angle but will correct any error by adding it onto the next turn it performs. An example of an inaccurate turn is shown in figure 3.

Figure 3: Pioneer robot after performing a turn. This angle was meant to be approximately 90 degrees but with up to 5 degrees of error.

The robot will continue to map until it runs out of squares that have not yet been visited. Below is an image of the robot after it has finished mapping and a screen shot of map produced by the robot. Compare the screen shot in figure 5 with the image of the initial set-up in figure 11.

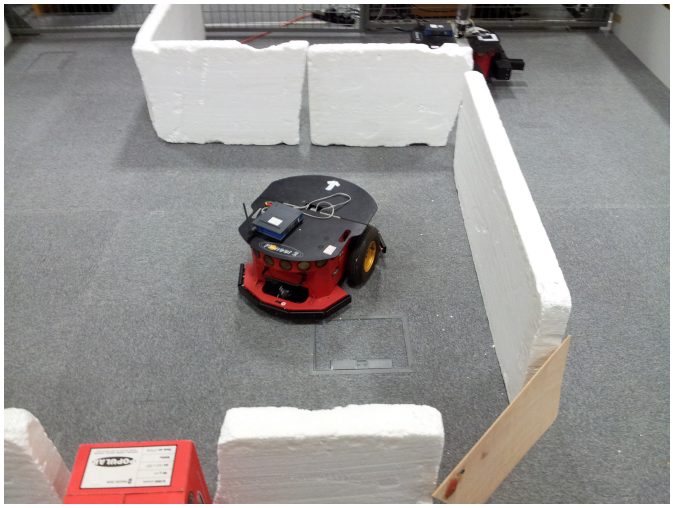

Figure 4: Pioneer robot after it has visited every part of the environment and finished mapping.


Figure 5: Screen shot of the map produced by the trial run shown in the images.

### 3.2 Hiding the Robot

Hiding the robot first requires the robot to localize within the map. Unfortunately, because the localization method depends upon the robot being placed in such a way that the robot lines up exactly with the previous grid, I could only get the robot to localize with the map if it started in the same square as in the previous trial run.

Figure 6: Pioneer trying to localize at start point in first environment.

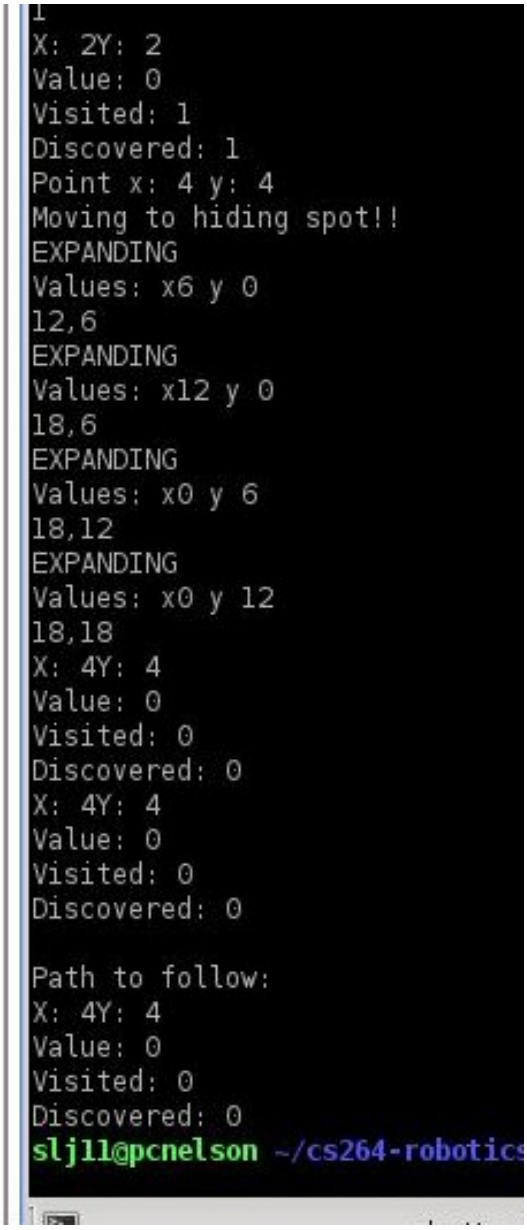


Figure 7: Screen shot showing a pioneer correctly localizing in a map where the robot starts at the same square (Note that save maps are buffered with 2 cells of padding hence why the robot changes position from (2,2) to (4,4). The hiding spot at the bottom is also (4,4) so the robot does not move.

However when I attempted to localize the robot in a slightly different environment (that I had also mapped previously), it did manage to localize the robot in squares outside of the initial start point showing that the concept worked but because of the requirement for the robot to sync up with the pre-defined grid it just wasn't terribly robust.

Figure 8: A Pioneer successfully localizing in another environment. The initial mapping ran from one cell behind (next to the wooden table)

Once the robot had localized, it should of picked a hiding spot from the available squares in the map and then moved to it. The localization attempt in the first example worked even though the robot started in the same place, but it just happened to chose the same cell as it started in as the cell to hide at. This seems logical as in the first trial run the robot started in a tight corner space. When the robot localized in the second environment it happened to choose a hiding spot outside of the mapped environment. This was due to inaccuracies in the sensor readings and using the wrong threshold. Unfortunately I could not fix this issue in time to attempt another run on the Pioneers but believe that the bug was a simple coding error.

### 3.3 Finding Map Differences

The final part of the assignment was to identify differences in a map of the environment. To test this I used the first environment with a human obstacle in place of one of the squares so as to create the illusion of a hiding robot.

Figure 9: A Pioneer about to find differences in the map. Complete with hiding human.

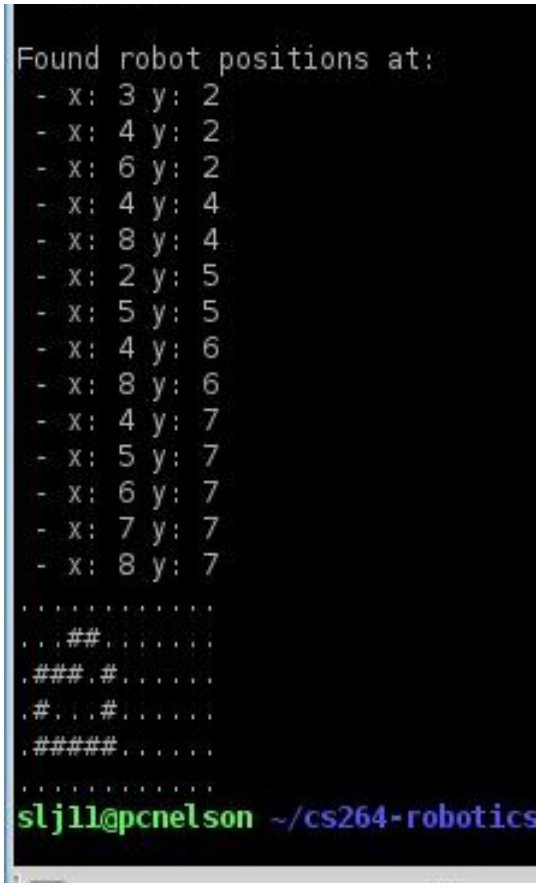


Figure 10: Screen shot after mapping the above environment. Notice the grid differs by one square on the map at the bottom because the robot has recognised the human obstacle.

Figure 10 shows the output of attempting to find points that are different in the map. While you can see that the grid is output with the correct differences at the bottom (compared with screen shot 5, the number of differences found does not reflect this. I believe this is down to a bug where the map isn't thresholded correctly or the two maps are offset slightly different. Unfortunately I did not have time to fix this bug to get the correct point outputting.

# 4 Discussion of Trial Runs

The trails runs shown in this document could have been a lot better. On the whole I found my robot to be relatively unsuccessful. However the ideas that underpin the design of the programs is still decent, but is just not smart enough to deal with the complexity and uncertainty of the real world.

For example, the least successful part of the trials was the localization function. This function would work if the robot lined up with the grid in exactly the same way as with the reference map but would fail otherwise. This is because the localization function made no attempt to generalize readings when matching the new grid to the old one. This could be improved by finding a way to compare the current and past readings that does not rely on exact matches with the grid, potentially by using probability to measure how closely new readings match with the reference map and choosing the position that matches most closely after several readings over time.

The mapping of the environment was the part of my application which worked the best. However there were still a few bugs. Firstly, the map maker program only really works when the environment takes the form of taxicab geometry due to the nature of the occupancy grid and the way the robot moves between cells. Mapping was virtually impossible with diagonally shaped or curved walls. Secondly, my cell size was set to use 60x60cm cells which would fit a robot but was too small to accommodate the robot's turning circle meaning it occasionally crashed into the

walls when turning. Making the size of the cells larger would of fixed this issue but would also of decreased the overall resolution of the map.

Both robot hiding and difference detection worked moderately well and had good ideas driving them, however they failed due to some weak coding that wasn't robust enough for the real world. If I'd of had more time to tinker with the robot I believe I could have fixed these errors to give some decent results and would of got the robot to correctly move to a hiding spot and find a hidden robot in the environment using my existing implementation.

# 5 Evaluation of Project

In conclusion I strongly feel that I approached this assignment from the wrong perspective. If I had to write it again from scratch I would of taken a more reactive approach to the movement and mapping of the robot instead of relying on having to move on a cell by cell basis using taxicab geometry. The simple random walk that I used in an earlier assignment was able to map the environment far quicker than the approach I used here, but lacked the ability to decide when it had finished mapping. Ideally I would have created a mapper that reactively moved around the environment but had a terminating condition and whose movement was weighted toward unexplored parts of the map.

I believe that this approach would have allowed me to move around the world quicker and with a great deal less error than with the current approach. While I was happy with the way that the robot moved from point to point and the correction to the movement over time by feeding any error back into the next movement, I still feel the movement system could of been better fine tuned using a time integral in the p-controller and better gains to achieve sharper movement, especially when turning.

Additionally, due to the fact that the amount a robot moves is guaranteed not to be perfect and that the internal measure of a position or angle given by the robot software may not even be correct I would of liked to of accounted for this in some way. This could of possibly been done by correcting measurements based on the certainty we have about the robots position according to factors such as how far the robot has moved from its initial position.

Localization proved to be the hardest part of the assignment for me. The approach that I took depended far too much on a perfect world and perfect robot positioning. It was also extremely computationally heavy with it having to compare all cells in two grids multiple times. To fix this I would of liked to of incorporated probability regarding potential points that my robot could be at and finding a way to generalize the shape of the new measurements to compare with old measurements. It is also worth noting that the approach I took only works if the robot is placed with the same orientation as the reference map. It would of been nice to look into solutions which could also handle multiple orientations.

One of the most positive parts of the assignment was the thresholding, which I found to generally work quite well. Otsu's method seemed to reliably separate wall cells and floor cells with decent accuracy. This was useful as it meant that erroneous sensor readings would be removed from the map and corrected for fairly reliably. A comparison of the values before and after are shown in figure 11.

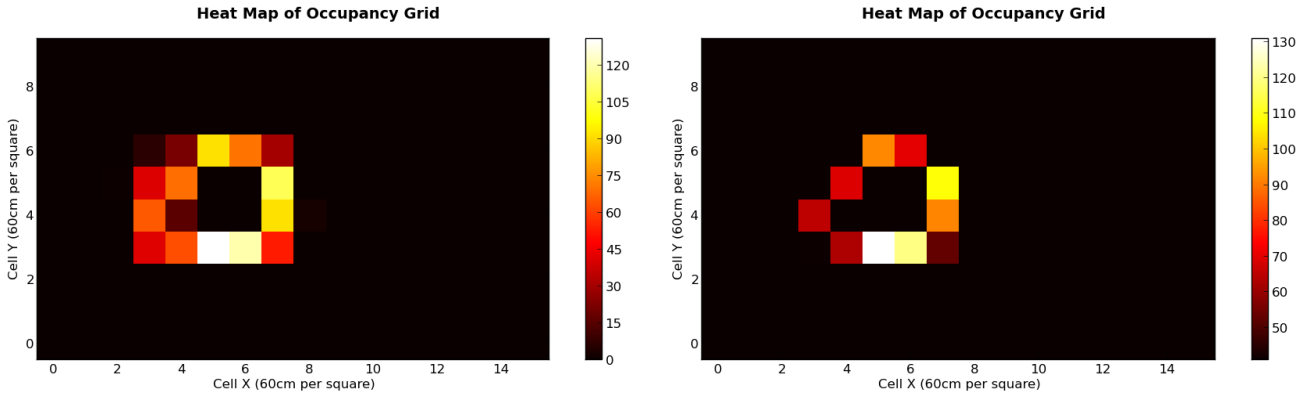


Figure 11: Shows the result of performing Otsu's method of thresholding on the occupancy grid built from the first trial run (output shown in figure 5). The left heatmap shows without thresholding, compared with the thresholded map on the right.

As mentioned before I believe that the techniques I have used to find hiding robots and to find and move to hiding spots are both pretty decent, however I believe that the supporting code was not robust enough or not well tested enough and therefore let me down on the real robots. I believe that they could of worked fairly well if I had more time to iron out the bugs in this code.

A further point that I would consider changing is how I am using A* to find the shortest path between where I am now and where I wish to hide. At the moment the robot will find the shortest path and blindly follow it to the goal. If I were to write the program again I would still use A* to find the shortest path but perform checks to ensure that the path is still valid as I move along it and has not changed since the map was created. If something has changed (such as another robot hiding in the environment) I could update the map and perform another A* to find an alternate route round new obstacles.

The biggest lesson learnt from this assignment was that the only perfect model of the world is itself. There are far too many variables in the real world that do not appear in the simulator and therefore cannot be planned for without sufficient testing on real robots. I found that I spent too much time attempting to get highly accurate movement (particularly with turning) when I should have accepted that moderately accurate movement can still be used to achieve decent results. This would have given me more time to work on other features of the assignment and would of resulted in better trial runs.

In summary, I found this assignment to be one of the most challenging but enjoyable that I have worked on. The design and implementation of robotic controllers is a skill that can only be developed through experience, trial and error, and some clever ideas. I believe that the real world is the only decent measure of controller performance and that simulation is no substitute.