IAR Task 3 Final Report

Angus Pearson – s1311631
Jevgenij Zubovskij – s1346981

November 17, 2016

Abstract

Given a *Khepera II* robot with no prior knowledge of it's environment, we attempt to simultaneously localise and map, under the assumption that the world is static. We find the chosen methods incapable of reliably completing the task due to environmental and algorithmic factors (§5). This system is built atop stochastic *Adaptive Monte Carlo Localisation* [3] to correct compound localisation error, and a dynamical *Occupancy Grid* generation architecture, the combination of which constitute a SLAM system. The task requirement of food collection behaviour is implemented using an *A* Search* planner.

Our existing Architecture from the previous IAR Assignments [1][2] is further extended, building upon the reactive behaviour, Redis datastore and ROS/Rviz real-time visualisation.

1 Dynamical Occupancy Grid

We build up a map (occupancy grid) of the environment as the robot explores, where at time 0, the map is empty. In this implementation, the dimensions of the grid are effectively infinite, as there are mechanisms allowing for automatic expansion in the event of the robot leaving the soft boundaries of the grid. The grid is stored as a hashmap in Redis, not a 2D array, with missing keys being assumed unoccupied. This allows for rapid querying of arbitrary cells' occupancy, and expanding the grid is trivial given no keys need be updated.

However, a grid resize effectively invalidates any 2D cached array version of the map, all clients would be required to re-sync with Redis. To mitigate this, the grid's initial soft-dimensions are large, as a cache reload is time-expensive.

The Occupancy grid is probabilistic, meaning each cell's occupancy is an integer in the range [0..100], with 0 indicating certainly unoccupied, and 100 indicating a certain occupation.

1.1 Ray Tracing

As the *Khepera's* IR Range sensors give us the distance between the sensor and an object, we can first transform this information from the robot's reference frame to the global (map) reference frame, using a *Transformation Matrix*:

$$\begin{pmatrix} x_{map} \\ y_{map} \end{pmatrix} = \begin{pmatrix} cos\Theta & sin\Theta \\ -sin\Theta & cos\Theta \end{pmatrix} \cdot \begin{pmatrix} x_{robot} \\ y_{robot} \end{pmatrix} \quad (1)$$

Then using *Bresenham's Line Algorithm* [6] we can raytrace between the robot's position and the cell that we are seeing an object in. All the cells on this ray are updated – If we've never seen before, they're marked as unoccupied, or if they are already marked in the map, the cell's value is kept unless the new value is more strongly occupied. Backing off from a certain occupancy helps remove noise from the grid though must be done conservatively to allow the localisation algorithm to make corrections. An occupancy grid of the task environment evolved by this process is shown in Figure 1.

Not quite clear – cells between robot & object are marked unoccupied and cell with objec in is updated how?

Grid population is done asynchronously from the main control loop, on an event-driven basis – map updates are made as the control loop *publishes* pose and sensory updates to Redis.

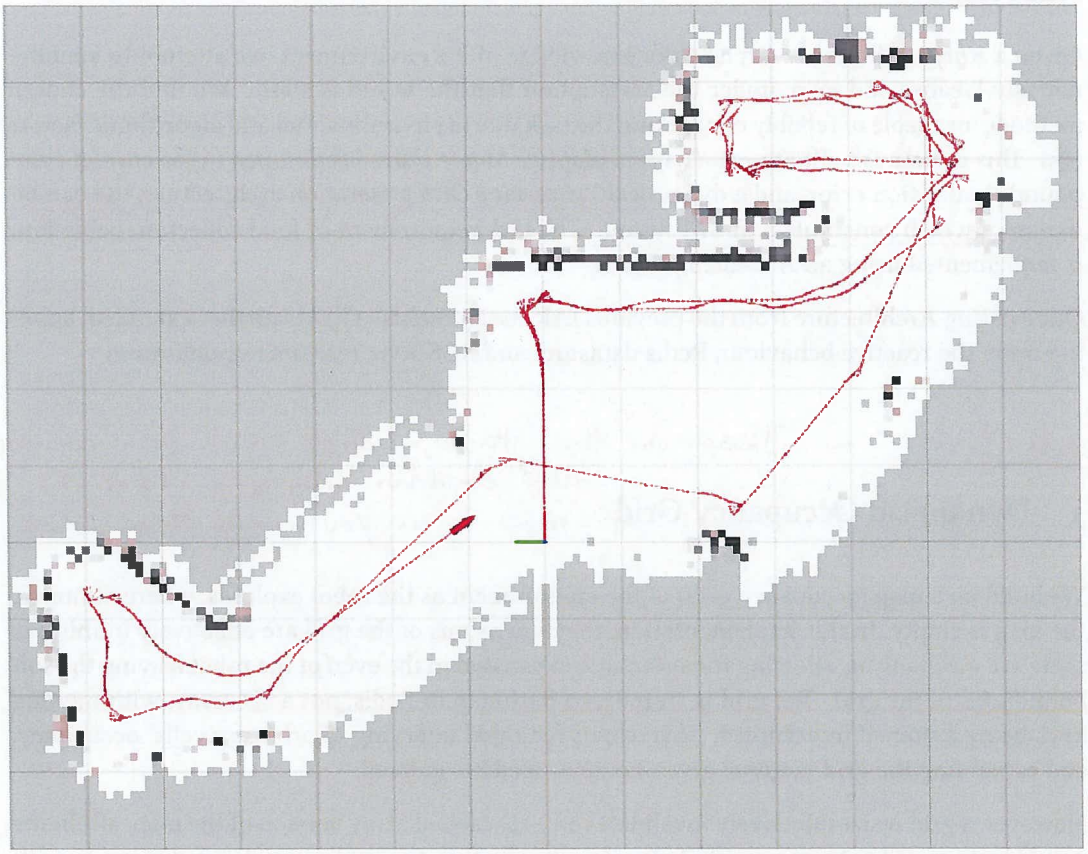

Figure 1: Occupancy grid, with overlaid pose (large red arrow) and Odometry Trail (smaller magenta arrows). Black indicates occupancy, white free space. Grey regions are not present in the Occupancy Grid.

Note that for this method to build a stable map, need to know pose of the robot.

2 Adaptive Monte-Carlo Localisation

The existing system carried forward from *Task 2* [2] maintained a pose estimate that at each control loop epoch was updated by a discrete sensor model for the *Khepera's* odometry. This model has no facility to model the compound error inherent to dead-reckoning with odometry and as the robot continues to move in it's environment the belief about where the robot is quickly diverges from reality. As the task requires multiple excursions and returns to within

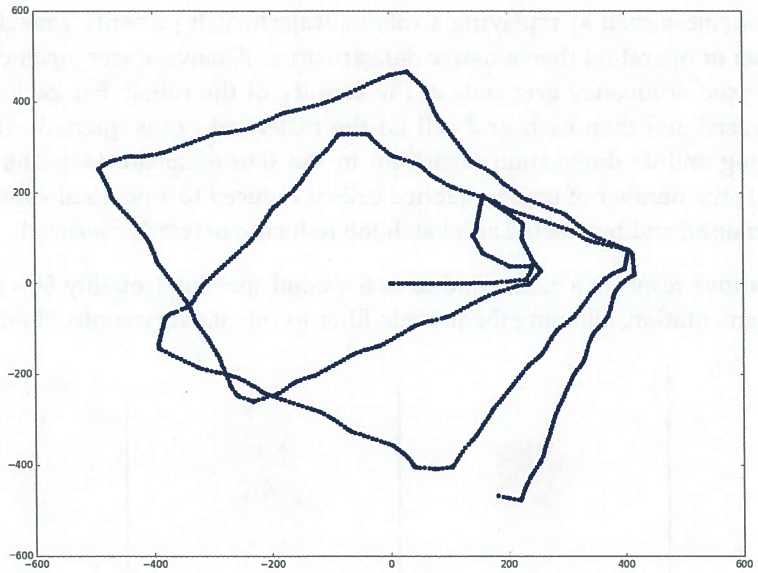

Figure 2: Illustration of Odometry error. The robot was following the border of a simple square environment, but the trace of it's path shows a spiral instead. Axes are (x, y) coordinates in mm.

10cm of the nest, this simple method alone is ineffective for localising the robot. This inherent drift is illustrated by Figure 2.

The *Adaptive Monte-Carlo Localisation* Algorithm as outlined in *Probabilistic Robotics* [4] provides a stochastic method of correcting for compound error and localising on an Occupancy Grid:

for a fixed (or converging) map?

Algorithm AMCL(X_{t-1}, u_t, z_t):

$\bar{X}_t = X_t = \emptyset$
for $m = 1$ **to** M **do**
  $x_t^{[m]} = \text{motion\_update}(u_t, x_{(t-1)}^{[m]})$
  $w_t^{[m]} = \text{sensor\_update}(z_t, x_t^{[m]})$
  $\bar{X}_t = \bar{X}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
end for
for $m = 1$ **to** M **do**
  draw $x_t^{[m]}$ from $\bar{X}_t$ with probability $\propto w_t^{[m]}$
  $X_t = X_t + x_t^{[m]}$
end for
return X_t

clarify the motion model used, and how query below is transformed into the weighting.

Particles, which are Normally distributed samples of the Probability Density function of the robot's pose space, each representing a hypothesis about the robot's pose are drawn from the prior set of particles X_{t-1} of size M using the *Weighted Reservoir Sampling Algorithm* [5], the effect of which is that particles with greater weight $w_t^{[m]}$ (i.e. the more probable hypotheses) have a greater chance of persisting once or multiple times in the new set of particles X_t. Note that the number of particles remains the same at all times.

100 particles are used in this implementation; The compute time for the *AMCL* grows quicky as the number of particles increases. Further to this, while Redis affords flexibility, data per-

sistence and fanciness such as replaying a robot's trajectory, it presents a much larger overhead per lookup or operation than a native datastructure. A naive *sensor_update* function will query around 7500 occupancy grid cells in the vicinity of the robot: For each particle, each sensor is raytraced and then each grid cell on the rasterised ray is queried. Using a rather cool pre-fetching and de-duplication algorithm in the sensor update (`mapping.py, Particles.update()`) the number of unique queried cells is reduced to a nominal value of 54. These queries are serialised and transacted as a batch job reducing network overhead.

These optimisations result in a 3.8 second to 0.08 second speedup (roughly $50\times$) compared to the naive implementation, allowing the particle filter to run at every control loop cycle.

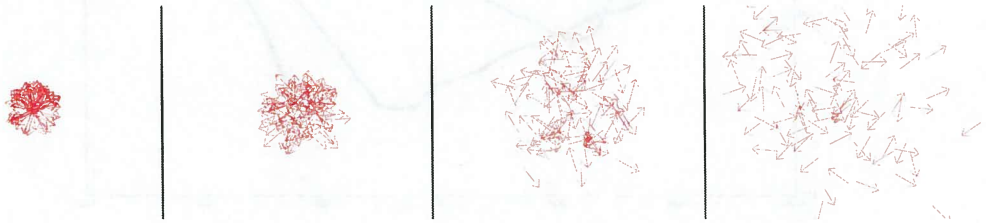

Figure 3: Dispertion of Particles by Monte Carlo method, from left to right with increasing time and a particularly high gain (i.e. each epoch disperses the particles greatly, assuming very large error)

Figure 3 illustrates how particles progressively diverge from the initial hypothesis, X_0. In our case, dynamical mapping affords a completely certain initial hypothesis given the co-ordinate system is relative to said starting position; This prevents the need to evenly distribute the particles over the environment at startup, so confidence in early localisations is high with no multimodal properties.

3 Planning

but in practice was not?

With SLAM we can assume that our localisation is accurate and proceed to the primary goal – gathering the maximum amount of food per unit time. For that we need to be able to plan out actions, which involves calculating routes to navigate the world and sequencing them accordingly.

3.1 Path Planning Space

(Path) Planning Space is a term used in this report to denote the representation of the physical world as the planner sees it, in this case (x, y, Θ). The existing use of this pose representation from Task 1 & 2 informs the choice of *Path Planning Algorithm (§3.2)*.

3.1.1 Method Chosen

The main methods of for planning over this representation are: [7]

- *Occupancy Grid*
- *Graph (Topological Map)*

The difference between these two from the perspective of planning is the efficiency of algorithm working on the data and the pre-processing as it goes into the algorithm. Because out Occupancy Grid is stored on a *Redis Server* and it has very fine resolution (aka granularity), the number of grid squares (referred to as *Grid Cells*) is immense. To obtain a graph representation of the planning space at each epoch for the map at the resolution of the grid as implemented is intractable from a compute time perspective.

Therefore planning over the Occupancy Grid is selected in preference to an abstracted representation such as a topological map.

3.1.2 Alterations

The granularity of the *Planning Grid (Planning Space)* did not necessarily have to match that of the occupancy grid on Redis for the following reasons:

- Robot Dimensions - the robot's dimensions are larger than the mapping (server-side) granularity
- Odometry Distortion - extremely finely planned path on an occupancy grid introduces many turns, increasing odometry drift rate considerably [2]
- Runtime Optimization - a coarser grid reduces the depth of a given plan over the same route, whilst sacrificing physical parity of said plan

The planning granularity is configurable though by default takes the same resolution as the Occupancy Grid.

Moreover, the *(Grid) Cells* store their actual (x, y) coordinates as well as occupancy - whether they are free space or not. Grid cells are stored in a two dimensional array which is indexed into using actual (x, y) coordinates. Furthermore, said array is also dynamically expandable in both dimensions so as to accomodate any number of cells (limited only by memory capacity) in order to not miss a valid optimal path due to any array bounds which also makes this solution more extensible and easily modifiable.

A local cache of the Occupancy Grid to make planning more performant. As already noted calling out to the Redis server at a high rate incurrs a large overhead - a cached version can be used to plan without concern for access frequency.

3.2 Path Planning Algorithm

This method should allow the robot to navigate the mapped space between any two unoccupied *Cells §5.1* of the planning space.

3.2.1 Research

Firstly, there are many options for the *Path Planning Algorithm* that search for a path between a goal and a destination location:

- *Floyd-Warshall Algorithm* [8] - a many-to-many algorithm for planning routes
- *Djikstra Graph Search Algorithm* [9] - a one-to-many algorithm

- *A* Search Algorithm* [10] - a one-to-one algorithm

These are some of the most common and robust algorihtms used in path planning. However, even based on the information above it can be concluded that Floyd-Warshall algorithm is not what we were looking because we have a single starting point - the position of the robot. Moreover, empirical data [11] shows that the best-first search algorithm is considerably faster than the Djikstra algorithm and in fact is in general considered one of the fastest planning algorithms – it's variations as well as it's purest form are used industry-wide in robotics. More reasons for choosing A* are outlined in *§3.2.3*

3.2.2 Method Chosen

A* was chosen for path calculation to the target grid cell. The pseudocode explaining its basic operation is as follows:

../../astar_pseudo.py

```
# add starting cell to open heap queue
heapq.heappush(self.opened, (self.start.f, self.start))

while len(self.opened):

    # pop cell from heap queue
    f, cell = heapq.heappop(self.opened)
    # add cell to closed list so we don't process it twice
    self.closed.add(cell)
    # if ending cell, return found path
    if cell is self.end:
        return self.get_path()
    # get adjacent cells for cell
    adj_cells = self.get_adjacent_cells(cell)
    for adj_cell in adj_cells:
        if adj_cell.reachable and adj_cell not in self.closed:
            if (adj_cell.f, adj_cell) in self.opened:
                # if adj cell in open list, check if current path better
                # than the previosu one for this adjacent cell.
                if adj_cell.g > cell.g + self.cell_distance(adj_cell, cell):
                    self.update_cell(adj_cell, cell)
                else:
                    self.update_cell(adj_cell, cell)
                    # add adj cell to open list
                    heapq.heappush(self.opened, (adj_cell.f, adj_cell))
```

Figure 4: The A* algorithm pseudocode[12] adapted to use the *Planning Space* chosen - an occupancy grid

../../astar_path_reconstruction.py

```
cell = self.end
path = [Cell(cell.x, cell.y, True)]

while cell.parent is not self.start:
    cell = cell.parent
    path.append(Cell(cell.x, cell.y, True))

cell = self.start
path.append(Cell(cell.x, cell.y, True))

path.reverse()
```

Figure 5: The A* algorithm path reconstruction pseudocode[14] adapted to use the *Planning Space* chosen - an occupancy grid

3.2.3 Alterations

The chosen *Path Planning Algorithm - A** (§3.2) uses the movement and *Heuristic Cost* to estimate the total cost of including cell into the claculated path. As we know the pose of each grid cell, *Euclidian Distances* were chosen to be a measure of transition cost between cells making it the cost function betwen any two cells A and B. The *Heuristic Function (Cost)* is 10 times the Euclidian distance from currently considered cell to the goal. The reason for such a high number is that we wish to see the algorithm converge as fast as possible while finding an adequately short path to the desired destination which also reduces the amount of Redis Server requests for unknown occupancies. As a side note, the movement cost is simply the sum euclidian distances between nodes along the (considered) path involving said cell.

Secondly, the A* pseudocode had to be adapted to work on a grid instead of a graph which was solved by a simple function obtaining its neighbours by using the considered cell's pose to get its adjacent ones from the *§3.1 Occupancy Grid*.

Most iterations of the A* algorithm rely on either graphs [14] or only movements up, down, left or right [13]. However, it seemed like a wasted opportunity to not allow the robot to move diagonally. Such cells are referred to as *Diagonal Cells* - cells diagonal to the currently considered cell - whose adjacent cells we wish to consider for furhter pathing. The scenario where the diagonal cell is blocked by two of its neighbours must be accounted for. In this case the diagonal cell can no longer participate in path calculation as shown in the Figure 6

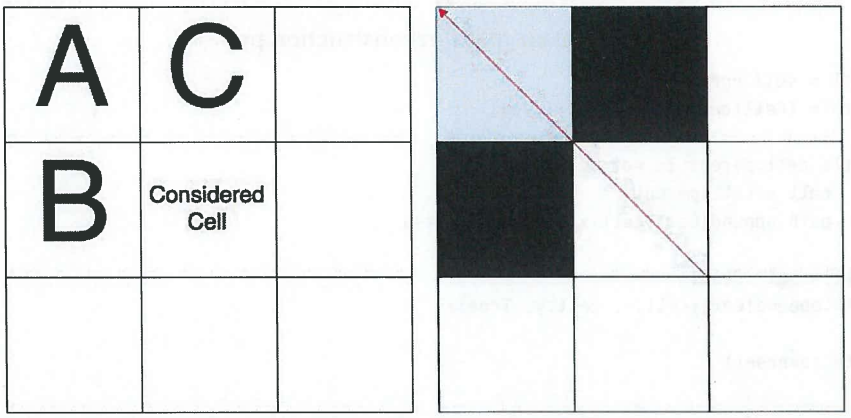

Figure 6: The **left** figure shows the *central* cell (square) is the so called *Considered Cell* whose neighbours we are trying to obtain to plan a path. The **left** figure shows that we cannot consider the *Diagonal Cell* A as a valid neighbour because it is blocked by B and C making the movement along the red arrow as seen in the **right** figure impossible

3.3 The Planner

This section is concerned with the intergation of the algorithm and data structures described above into the main control loop and its interaction with previosuly established subsystems [2]. The algorithm determining the next action or travel destination and sequencing the execution of said action is called the *Planner* and is, therefore, responsible for maximizing the food collection rate by sequencing events governing said rate.

3.3.1 Method Chosen

Now that we are able to localize ourselves, know the map and can calculate paths on the planning grid, we can set priorites for action to maximize food gathering. Firstly, it is important to note that each *Food Source* has only one unit of *Food*. Upon return to the *Nest* and dropping off currently collected food the planner resets the food sources which now have food units again, starting a new *Food Collection Round (a.k.a. Round)*. Action priorites were set as follows:

1. *Find* the first food source to begin periodic behavior in the following points
2. *Explore* once per Food Collection Round
3. *Collect* closest uncollected known / recorded food source
4. *Return* to the nest and drop off collected food

We aim to find some food source early to be able start collecting food as soon as possible. Once this is done, exploration for new food sources becomes less of a priority and is only invoked once per round as we do not wish to waste too much time on exploration when we already have a known location to collect from. Next on the list of priorities is collecting all the food before returning to the nest to drop it off.

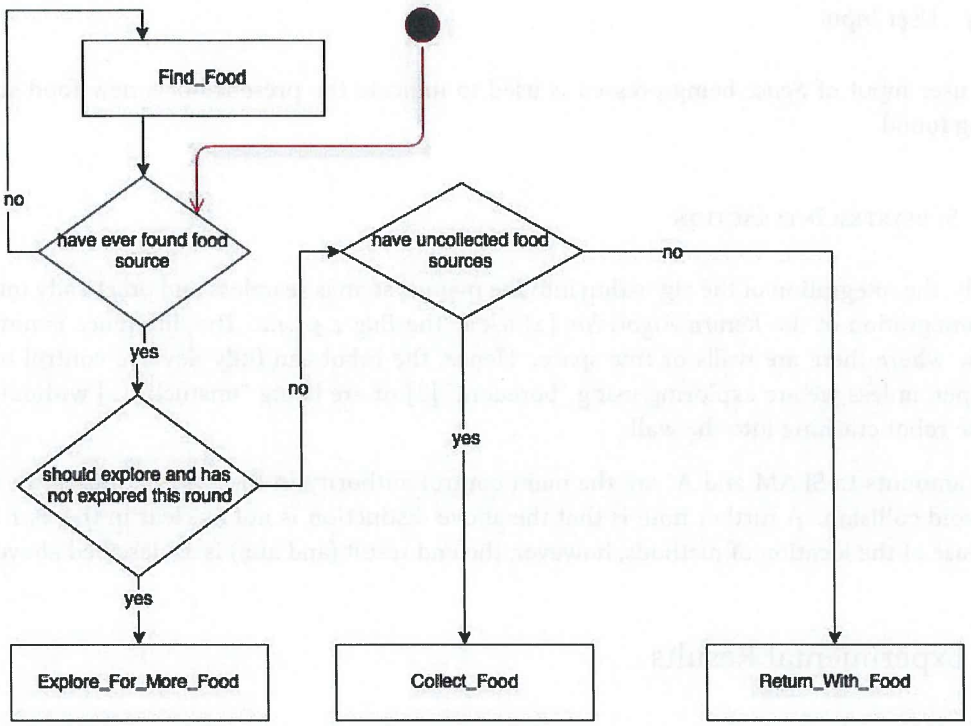

Figure 7: The planning algorithm - the *Planner* in control-flow form, this decision making tree is implored on every iteration of the control loop and determines the action the system is to follow.

The robot aims to collect the closest food first (based on Euclidian distances). The movement between cells is movement along the vector between their recorded coordinates by adjusting the angle along the *Vector of Approach*, same as *Return Algorithm*[2]. Moreover, sometimes unxecpected events can happen that coud lead the robot away from said approach vector. Hence, the next destination is decided (as per above priority list) when the following events happen:

- Food collected from a Food Source
- Food dropped off at nest - round finished
- Path lost - went a distance over the threshold away from the path

3.3.2 Alterations

After reviewing initial *Experimental Results* it was decided that due to the grid granularity being set (by default) in the same order of magnitude as robot dimensions, it is reasonable to assume that no two food sources can exist in the same grid cell as the cell sizes are assumed to be set by the user / programmer to a granularity not exceeding robot dimensions. Lastly, because the algorithm needs to find new food sources, the boredom mechanism [1] was reinstated into the control loop and is the exploration mechanism and is utilized during the *Find* stages of the algorithm.

3.3.3 User Input

The user input of *Space* being pressed is used to indicate the presence of a new food source being found.

3.4 Subsystem Interaction

Lastly, the integration of the algorithm into the main system is seamless and practically mimics the integration of the *Return Algorithm* [2] a.k.a. the *Bug 2.4.1.1.2.* The difference is now we know where there are walls or free space. Hence, the robot can fully devolve control to the planner, unless we are exploring using "boredom" [2] or are being "unstuck" [2] without fear of the robot crashing into the wall.

This amounts to SLAM and A* are the main control authority in the system, unless we need to avoid collision. A further note is that the above distinction is not as clear in the *§5.1 Code* because of the location of methods, however, the end result (and aim) is as descibed above.

4 Experimental Results

Each subsystem component was tested independently from the rest of the stack. Due to localisation failing to correct for compound error in odometry, the robot is unable to reliably complete the task in the environment.

4.1 Planning

The motion planner was tested with hardware in the loop, planning around virtual obstacles to navigate to a desired location, identify new food sources and take the shortest route between multiple food sources and the nest. The success rate for this was 9/10 trials, where the failure was due to compound odometry error.

Not very clear exactly how this was tested – what constituted a trial? What counted as success?

4.2 Mapping & Localisation

The actual mapping of occupancies according to localisation is correct, with sensed obstacles in the task environment being added to the grid in the correct position relative to the robot. ✓

Okay, I saw this in demonstration but would be nice to include more examples.

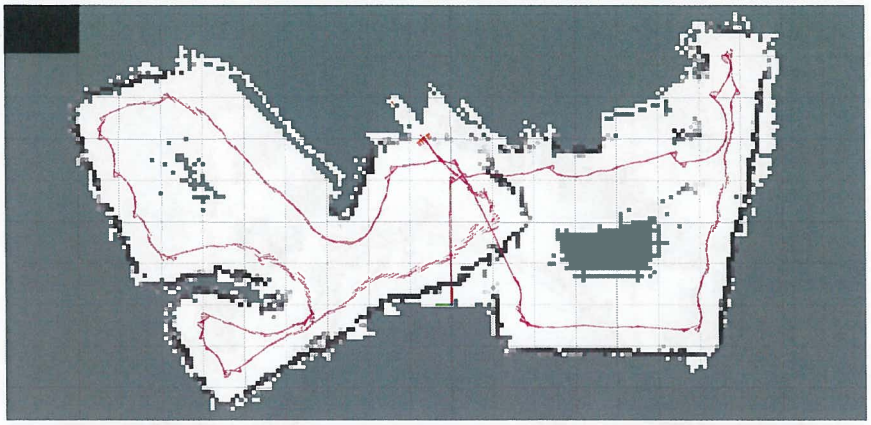

Figure 8: Invalid Occupancy Grid, showing a clear localisation error's resulting in translated and rotated shapes. The shapes are in the correct proportions.

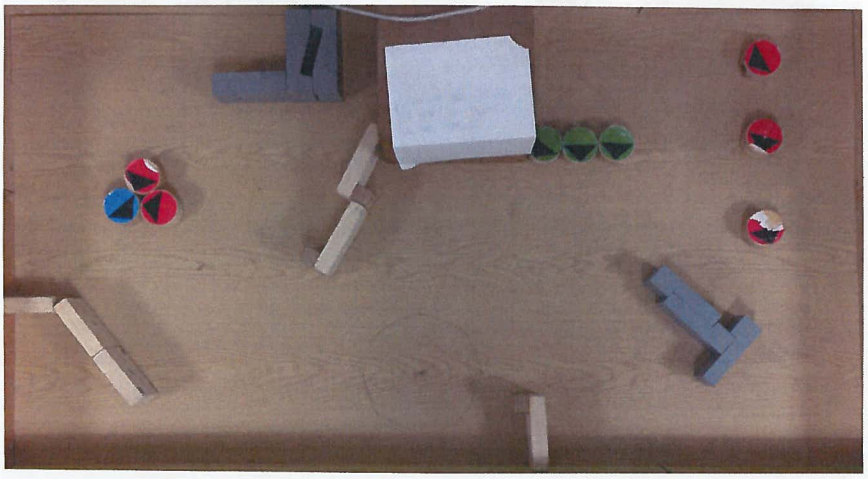

Figure 9: Top-down image of the arena as a comparison.

In a minimal environment, a compound error in localisation still appears, mainly comprised of a rotational error resulting in a corresponding rotation of the map as time progresses (Fig. 8). This test's resulting occupancy grid and odometry trace is shown in Figure 10

↳ not minimal environment

Did your mapping method include any loop closure?

It seems clear that for this task it would have been better to exploit the known map (could be refined by the robot as it moves) so that odometry could be corrected by identifiable map locations.

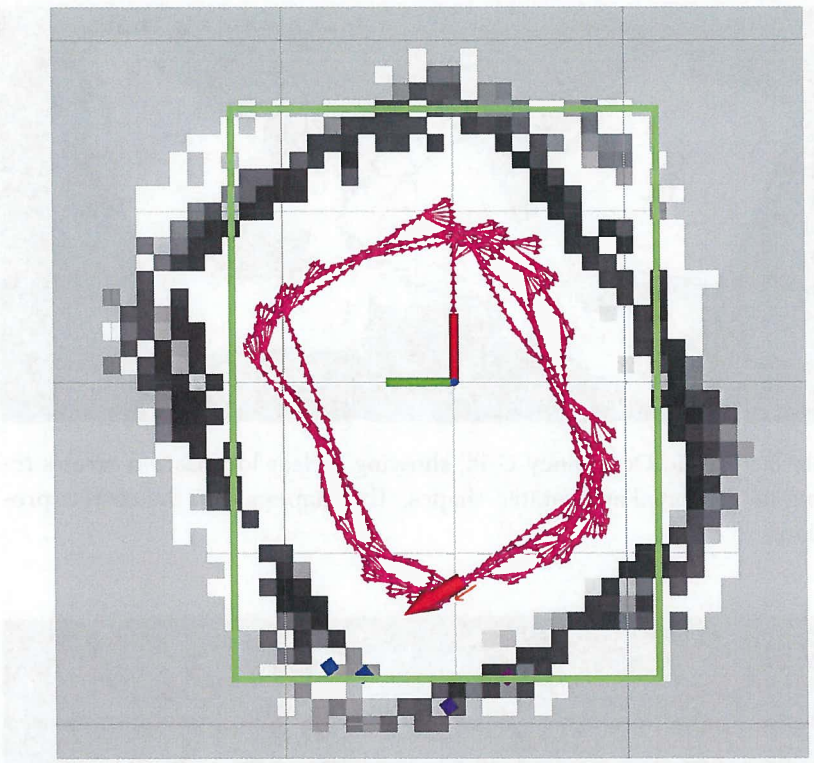

Figure 10: Map after following the inside of a simple rectangular environment, showing the slow rotational error in the grid. The environments actual boundaries are shown by the green rectangle, dimensions $20cm \times 30cm$. The Robot started at the origin (centre).

5 Conclusions

The Grid occupancy retention policy has a large effect on the rest of the SLAM stack. As the sensors do not always report the same distances as one-another, and that we snap to a discrete granular grid it is not uncommon for a cell previously marked occupied to either be overwritten completely or have it's occupancy certainty degraded.

So would it be better to change this?

In addition to this, if we first encounter a new obstacle after traversing a large open space of which there are many in the task environment, affording no prior features to relocalise on, this new feature will enter the map in an erroneous position. This error will then carry through to the robot's position estimate whenever it believes it can see this feature.

Further to this, the 'rough edged' Occupancy Grid map gives an already noisy feature for *AMCL* to attempt to localise on – meaning none of the hypotheses (particles) hold a high probability of being true to the real world, and many that should quicky be defeased remain in the state estimate. Adding more Gaussian noise to the *AMCL* resampling function in an attempt to alleviate these localisation errors results in teleportation, oscillation or positional drift induced by the filter itself. The whole system quicky deteriorates into an invalid state once this begins happening.

Planning verifiably worked atop correct localisation, devising and executing optimal plans. However, failure to localise renders it's attempts at completing the task moot, as the believed home location falls outside the 10cm maximum radius.

Not shown in results?

The basic problem with your approach is that AMCL requires a known map, and occupancy grid mapping requires known pose; their combination is not enough to constitute SLAM, as you need to track the joint uncertainty over pose + map.

References

[1] IAR 2016 Task1 Report
Angus Pearson, Jevgenij Zubovskij

[2] IAR 2016 Task2 Report
Angus Pearson, Jevgenij Zubovskij

[3] Principles of Robot Motion: Theory, Algorithms, and Implementation §2.1 'Bug Algorithms'
Howie Choset

[4] Probabilistic Robotics (Intelligent Robotics and Autonomous Agents), MIT Press, 2005
Thrun, Sebastian and Burgard, Wolfram and Fox, Dieter

[5] Reservoir Sampling, Dictionary of Algorithms and Data Structures
Black, Paul E. (26 January 2015)

[6] A Linear Algorithm for Incremental Digital Display of Circular Arcs
Commun. ACM, J Bresenham, Feb. 1977

[7] Map Representation Comparison and Planning (online article)
http://correll.cs.colorado.edu/?p=965

[8] Theory of Algorithms (lecture)
http://cs.winona.edu/lin/cs440/ch08-2.pdf

[9] Greedy Algorithms (online article)
http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/

[10] A* Comparison (online article)
http://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html

[11] Comparative Study of Path Planning Algorithms
http://research.ijcaonline.org/volume39/number5/pxc3877058.pdf

[12] A* Search Algorithm (online article)
http://web.mit.edu/eranki/www/tutorials/search/

[13] Introduction to A* (online article)
http://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html

[14] A* Search Alogorith (online article)
*https://en.wikipedia.org/wiki/A*_search_algorithm*

It is good that you have taken an ambitious approach and researched and implemented a range of relevant methods. However, taking advantage of information provided (a fixed map) would have produced a more effective solution to the task. The presentation of your approach in this report is also not always sufficiently clear.