Coursework 1 - MMSW 1

THE KIDNAPPED ROBOT PROBLEM

November 24, 2015

Aidan Scannell - 25%
Asher Winterson - 25%
David Mathias - 25%
Maxwell Martin - 25%

1 Introduction

This report will outline the approach that was developed for solving the kidnapped robot problem. This involved developing algorithms capable of localising a robot within a known environment but at an unknown position and moving it to a target location. This was achieved in simulation using the BotSim library in Matlab and then implemented onto a real robot. The overall procedure is shown below:

```
while Not Converged do
    Localisation Algorithm
end
while Not at Target Location do
    Route Planning Algorithm
    Movement
    Relocalisation
end
```

Algorithm 1: Overall

The robot was limited to using a single ultrasound sensor, an odometry sensor and two motors. The robot was designed and built to represent the simulated robot as accurately as possible. This was achieved by positioning the ultrasound sensor directly above the turning centre, located in the middle of the two motors. In theory this configuration enabled the robot to turn without displacing the sensor, although in reality this is unlikely.

The localisation algorithm utilised a particle filter to approximately calculate the robot's position. The Euclidean distance was used to score the particles with a low variance resampling procedure.

Once a suitable estimation for the position of the robot was calculated, a wave-front based route planner was implemented to generate a discretised route to the target. To ensure that the robot did not clash with a wall, a smaller, inwardly offset map was generated and used in the route planning.

The motion of the robot was divided into rotate and translate, therefore requiring separate turn and move commands. The list of unit moves from the route planner was then combined into turn and forward commands for the robot.

The simulated robot performed scans and relocalised along the route to ensure its accuracy as it moved.

2 Localisation

This section of the report will outline and discuss the localisation algorithm that was used. A particle filter method estimates a location of the robot using the following approach:

```
Initialisation
while Not Converged & Under Maximum Number of Iterations
do
    for All Particles do
        Score Particles
        Estimation of Position
        Estimation of Orientation
    end
    Resample
    Move
    Test Convergence
end
```

Algorithm 2: Localisation

2.1 Initialisation

The particle filter required particles with equal weights to be randomly generated within the map. A higher number of particles reduces the variance but has a greater computational cost. For this reason an optimal particle density was required that balanced computational time and variance. The chosen particle density was obtained through trial and error and was achieved dynamically for each map by considering the maps area. The polyarea function in Matlab was used to calculate the maps area and thus determine the number of particles required.

2.2 Number of Scans

The number of scans performed by the robot and particles determined how well the particles represented the robot. Too few scans resulted in many particles in close proximity to the robot returning high Euclidean distances and thus low weights. Increasing the number of scans improved the likelihood that a particle in close proximity to the robot would obtain low Euclidean distances and thus high weights. It also resulted in a more accurate estimate of each particle's orientation.

2.2.1 Real & Simulation

In the simulation twenty scans were used because it enabled particles in close proximity to the robot to obtain a high weight and select an accurate orientation. Increasing the number of scans is however more computationally expensive. In the case of the real robot there is also a delay between scans, introduced because the sensor is one-dimensional and requires mechanical turning to obtain all of the scans. The time constraint and the errors incurred from scans at 45°'s to walls resulted in four scans being selected for the real robot.

2.3 Scoring The Particles

The particles were scored using the Euclidean distance (a metric for the difference between each particle's scans and the robot's scans). Equation 1 shows the equation that was used, where d is the Euclidean distance, n is the number of scans, r_i represents a robot's scan and p represents a particle's scan. The Euclidean distance was calculated for every orientation for each particle and the minimum was selected. The orientation that achieved the minimum Euclidean distance was representative of the particle's orientation [2].

$$d = \sqrt{\sum_{i=1}^{n} (p_i - r)^2} \tag{1}$$

The weight of each particle was then calculated by taking the reciprocal of the Euclidean distance and normalising these values.

2.4 Resampling

Resampling too frequently can result in decreased diversity and result in the particles converging to the wrong point. Conversely, resampling too infrequently can lead to particles staying in positions of low probability (too diverse). This diversity becomes an approximation error, also known as variance of the estimator. As the variance of the particle filter decreases the variance of it as an estimator of the real belief increases [1]. For this reason a low variance resampling procedure was utilised. The resampling algorithm selects a single random number and uses this to select particles. The particles that are selected have probabilities that are proportional to the sample weight. This is achieved by selecting a random integer between zero and the number of particles being used. Algorithm 3 shows the resampling procedure based on Sebastian Thrun's low variance resampling that was used for the project. The selection utilises a sequential process which improves the procedure's efficiency and thus practical performance.

```
N = Number of particles
r = Random number between 0 and N
w = Weight
i = 1
c = $w_1$
for $m = 1 : N$ do
    u = r + $\frac{m-1}{N}$
    while $u > c$ do
        c = c + $w_i$
        i = i + 1
    end
    Update particle set
end
```

Algorithm 3: Resampling [1]

2.5 Movement

The movement of the robot is crucial for effective localisation. It is important that the scans are taken at locations that result in accurate sensor readings. Moving the robot effectively enables unrealistic particles to be moved off of the map and resampled to more likely positions.

The movement algorithm, on the first iteration, interrogates the scan to find the maximum distance to any wall and the corresponding angle. In subsequent iterations, steps of the maximum distance divided by an arbitrary amount (e.g. 10) are taken. This allows the robot to travel throughout the map in an exploratory fashion, which performs particularly well for maps with repeated features. To prevent the robot from repeatedly oscillating between equidistant points, a check is made every iteration and if a flag is raised the second largest distance is used. When turning, the direction of rotation with the least movement is chosen to ensure the movement noise is minimized.

Due to the random nature of the kidnapped robot problem, a common problem was collisions with the wall during localisation. During each scan, if a measurement is found to be below a threshold that is determined by the radius of the robot, then evasive action is taken by turning towards and moving into a location with greater space.

2.6 Convergence

During every iteration, it must be decided whether the particles have converged within a reasonable distance of one another to ultimately decide an estimated position for the robot. For this task a clustering approach was used. This allows for an undefined number of clusters to exist, therefore increasing the diversity of potential convergence points and allowing for faster convergence. The approach searches through the particle's positions and angles and groups all values within a defined tolerance together with a top level particle representing the cluster. Convergence is reached when a maximum particle density is reached. This threshold value is defined by the tolerance described above and the cluster density. Both of these are defined dynamically as a function of map area that has been found by performing linear regressions on various map areas (for the real robot these values are constant). The top level particle that is representing the converged cluster is chosen as the estimated position of the robot. The data is scaled for the orientation tolerance so that a single value can be used to define all of the x, y and

θ tolerances.

3 Route Planning & Movement

The route planning used the robot estimation outputted from the particle filter localisation and generates a discrete route based on unit moves in the eight cardinal directions (N, NE, E etc.). The route planner used was a variation of the wave-front algorithm that was based on the map being split into a grid. The route was then smoothed into turn and move commands for the robot to execute.

3.1 Minkowski Sum & Obstacle Inflation

A crucial part of robot motion is ensuring that the robot does not collide with obstacles. The simulated robot only has a position and no size, while obviously a real robot has a size envelope that cannot overlap with walls. The advantage of the simulated robot is that it can be treated as a point and as a result is far easier to handle for both route planning and movement.

The Minkowski sum allows a physical robot to be considered a point as it inflates the obstacles and wall by the size of the robot. In this scenario of the kidnapped robot, it means that the map the robot can move in is reduced so that it does not clash with a wall.

Figure 1 shows the default map with three different Minkowski sums added to it. The LEGO robot used had a radius of 11 cm and so the map was offset by its size.

Algorithm 4 outlines how the Minkowski sum function works. The first part is to test to see if the list of vertices is creating a clockwise or anti-clockwise polygon. This is required to ensure that the new map is offset in the right direction. The next step is to iterate through each vertex and calculate the angle between its adjacent edges. Depending on the size of the angle, the offset distance can change with more acute angles needing a larger offset. The angle is halved and used to create a vector equal to the length off the offset distance. Using this vector and starting at the old vertex it is possible to generate the new offset map vertex.

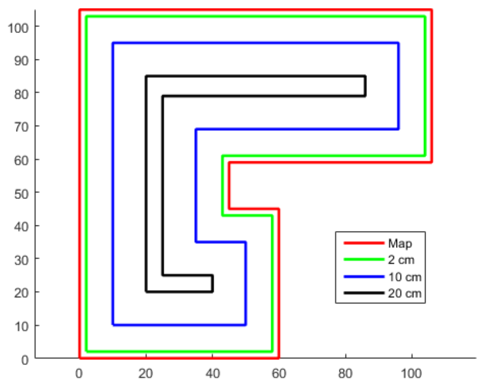

Figure 1: A plot of the normal map with three different Minkowski sums applied

```
Variable Initialisation
Test Polygon Direction
for each Map Vertex do
    Calculate angle between adjacent edges
    Calculate offset distance required from vertex
    if Polygon Direction > 0 then
        Offset direction = $-1$
    else
        Offset direction = 1
    end
    Halve angle
    Calculate translational vectors to convert old map to new
    Append new vertex in array
end
```

Algorithm 4: Minkowski Sum

This algorithm is very rudimentary as it cannot deal with self intersection or applying radii to corners with large internal angles. However, it is suitable for this application as the maps have relatively simply geometry and small offsets.

3.2 Route Planning

A wave-front based route planner was chosen as it was robust and offered simple moves for the robot to execute. The accuracy of this route planner is entirely dependent on the resolution of the grid used to discretise the map. A resolution of 1 cm was used as it was small enough to allow accurate routes at the <5 cm scale but not too small to be computationally slow. The grid was generated with the aid of the built-in BotSim function ‘pointInsideMap’. This process was performed outside of the main route planning function as once generated it would not change.

Algorithm 5 shows the main processes in the route planner. The planner takes the start point (estimate of robot position) and tests to see which adjacent grid positions are open. This then iterates across the whole map, cumulating the path cost and closing off positions previously visited. This propagates through the map like a wave front, hence the name. Once complete, the moves used are calculated from the target back to the start.

As this is moving in eight directions on a gird, the four diagonal moves will have a higher cost. A normal cost is equal to 1, while for diagonal moves it is equal to 1.414 ($=sqrt(1^2 + 1^2)$). This was needed to ensure that the lowest cost path was chosen.

If the robot estimate is inside the Minkowski sum (or off the map entirely) or the target is not in the map, then no path will be found. In this case the robot makes small movements to try and find the map.

One problem that had to be overcome was the route planner’s choice of diagonals that cut across internal map corners. This originated from the grid layout. To stop this happening, if the grid position in question was on a map edge, diagonal moves were bypassed.

The disadvantage of this route planner is that the robot can only turn in 45 degree increments and as result cannot move efficiently along different path angles. Also due to the grid based planning the function has to perform many iterations. However the main advantages of this approach are the ease of implementation and the straightforward turn and rotate moves generated for the robot.

```
Variable Initialisation
Generate Map Grid
Convert Real World Start and Target to Map Grid coordinates
while Target is not Found & Resign is not True do
    if No path options then
        Resign = True
    else
        Sort available grid positions by cost
        Remove lowest cost position and assign it to current
        position
        if Target is Found then
            Target = Found
        else
            for each of Cardinal Moves do
                New position = current position + move
                if New position is in map bounds & Not closed
                then
                    Calculate new cost
                    Add new position and cost to open array
                    Close new position
                end
            end
        end
    end
end
if Resign is not True then
    while Start is not reached do
        Work backwards to find specific route
    end
end
```

Algorithm 5: Wave-Front Route Planning

3.3 Movement Commands

The 'moves' array that is output from the path planning is iterated through to complete the required rotations and movements. Rotations are conducted by taking into account the previous rotation and the intended direction to find the best direction to turn. This minimises error by not making unnecessarily large rotations.

After each rotation the robot would then perform a movement. As the 'moves' array provides 1 cm moves, it was important to sum the distance if the same number was repeated to allow for a smoother movement. A 'while' loop was used to sum identical consecutive numbers. This 'sum moves' variable can then be multiplied by the Tacholimit value for 1 cm, allowing the robot to move forward the required distance in one step.

The simulation version followed an identical method but had the advantage of being able to specify non-integer numbers for movement or rotation. The real robot's TachoLimit values must be integers or the function will not work, this allows for rounding errors that will give a slight inaccuracy.

4 Sensor and Motor Noise Modelling

After a few practice tests using the motors and sensors it was soon obvious that both incorporated a level of noise. By quantifying this noise, the probability of the robot doing the required move or scan can be used to allow for a level of uncertainty when moving the particles that estimate the robot position. It was important to get a large enough sample size when testing to provide meaningful results that could be used to calculate the standard deviation. Figure 2 shows the standard deviation calculated for movement.

The standard deviation was found for forward and sideways movement and was found to vary across the distances tested. It was therefore not acceptable to use a fixed value. The standard deviation varied depending on the movement. It was noted that as the robot started to move longer distances, it began to drift further from its intended path and off to the side. From this plot it is also possible to obtain the optimal distance to move that will result in the least amount of noise.

Similar testing was done for the ultrasonic sensor noise and rotation. The sensor was found to be very accurate when measuring perpendicular to a wall. Problems began to arise when measuring at an angle. At 45 degrees the standard deviation grew to be very large, especially at close or far distances from the wall. This further reinforced our decision to use four scans as appose to eight due to the scans always being perpendicular to the walls. This dramatically increased the scan's reliability and the standard deviation was kept far smaller. Rotational noise was noticeably varied, as would be expected for larger rotations and resulted in a higher standard deviation.

References

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

[2] F Viani, P Rocca, G Oliveri, Daniele Trinchero, and A Massa. Localization, tracking, and imaging of targets in wireless sensor networks: An invited review. *Radio Science*, 46(5), 2011.

5 Appendix

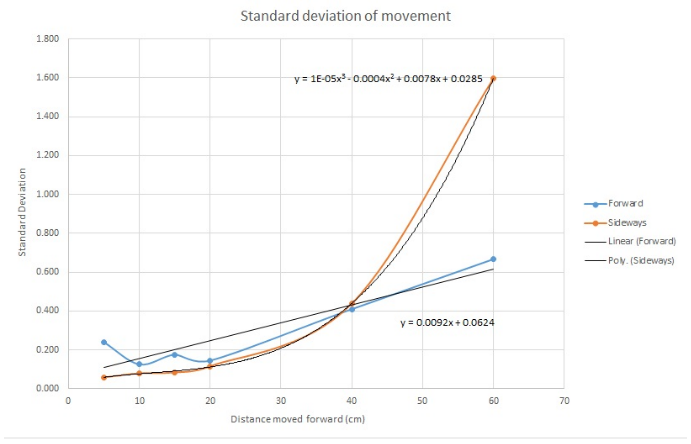

Figure 2: Standard deviation of robot movement displayed as a function of distance

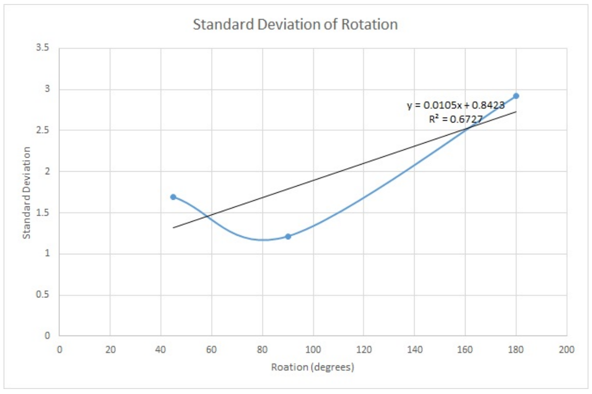

Figure 3: Standard deviation of robot rotation displayed as a function of rotation

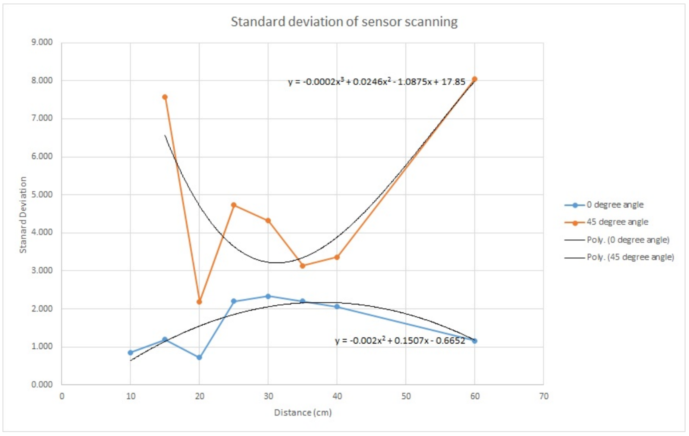

Figure 4: Standard deviation of sensor scanning displayed as a function of distance scanned