An Autonomous Meeting Assistant

951926 1001231 1024072 1028907

***Abstract*—In this report, we present a system which performs the 1996 AAAI Mobile Robotics Competition task "Call a meeting"[?]. The robot was tasked with bringing a certain number of people to a meeting room previously determined as available, all within a 15 minute time limit. To complete the task, we use probabilistic roadmapping, Monte Carlo localisation and a face detection module using the ROS framework, running on a Pioneer 3-DX platform modified with a tripod-mounted Kinect.**

I. Introduction

Solutions to complex tasks often require the use of multiple techniques to solve different sub-problems, and this task is no exception. We were required to implement a localisation algorithm to determine the position of the robot, a probabilistic roadmap to allow paths to be planned through the space, a navigation algorithm for path planning, a method of exploring the space, and some way of detecting people. We then had to implement a system which would combine all of these separate modules into a single system that would complete the task. In the next section we present some background information about techniques that are often used to solve these sorts of problems. We then provide a detailed description of our system and evaluate its performance. Finally, we discuss the experimental results and suggest areas for improvement.

II. Background

In this section, we discuss briefly the areas of robotics that are relevant to solving the task that has been set. We mention some specific algorithms and techniques that we might consider using, as well as describing briefly the benefits and drawbacks of some of the techniques.

A. Localisation

The aim of localisation is to obtain an estimate of the position of a mobile robot using sensor data [**?**]. While it is possible to use odometry data to do so, this data is inherently noisy and prone to error. In particular, odometry errors can arise from changes in the surface being traversed and the robot's weight, among others. An important point to note is that localisation can be performed with a prepared map, or generating a map in real-time. The much more complex latter problem is called simultaneous localisation and mapping, which we will not discuss here. See [**?**] for an introduction to the SLAM problem.

1) Bayes Filter: Many advanced localisation techniques are based on the Bayes filter, which uses a belief distribution $bel(x_t)$ to represent the state x_t. The calculation of $bel(x_t)$ at time t is dependent on $bel(x_{t-1})$ at time $t-1$, the last action u_t, and the last measurement z_t. In the first step, called the prediction step, the prior belief $\overline{bel}(x_t)$ is calculated. This step merges two probability distributions; the prior belief over the previous state x_{t-1}, and the probability of transitioning from that state to the posterior state x_t given that the action u_t was taken. This step does not take into account any measurement taken in the posterior state, predicting based solely on the knowledge of the action taken. In the second step, the measurement update, the posterior belief is calculated by multiplying the prior belief with the probability of being in the posterior state given that the measurement z_t was observed. The result of this multiplication is generally not a probability, and therefore requires normalisation using some constant α. As there are usually multiple posterior states, x_t is usually a state vector rather than a single state, and so the two steps will be applied multiple times in order to update the belief for each state being considered. The filter is recursive, requiring some idea of the initial belief $bel(x_0)$ at time $t = 0$. The initial belief is either a distribution centred on x_0, in the case where the initial position is known, or a uniform distribution over the space otherwise. As the Bayes filter is not restricted to finite state spaces, it is not possible to implement it for anything other than very simple problems. There are two families of algorithms for localisation, known as *recursive state estimators*, with various properties that permit the use of the Bayes filter in more complex estimation tasks [**?**].

Algorithm 1 Bayes filter [**?**]

```
1: Algorithm Bayes_filter($bel(x_{t-1}), u_t, z_t$)
2: for all $x_t$ do
3:     $\overline{bel}(x_t) = \int P(x_t \mid u_t, x_{t-1}) bel(x_{t-1}) dx_{t-1}$
4:     $bel(x_t) = \alpha P(x_t \mid z_t) \overline{bel}(x_t)$
5: end for
6: return $bel(x_t)$
```

2) Gaussian Filters: The basic principle of the family of Gaussian filters is the use of multivariate normal distributions, which can be formulated from a mean μ and covariance Σ, to represent belief. As a result, the assumption that the system is a linear Gaussian system is made; the initial belief must be a Gaussian, and both the state transition function and measurement probability must be linear functions. Although Gaussian filters can be extended to non-linear systems, they perform best when the system meets the assumptions made. One of the main advantages of such filters is the computational complexity, which is polynomial with respect to the dimensionality of the state space. The main disadvantage is that Gaussians are unimodal and therefore cannot represent situations in which there are multiple hypotheses; a situation that is often encountered in robotics. Examples include the Kalman filter and the information filter, which are derived from two different ways of representing Gaussians. Both of these filters can be extended to non-linear systems by using a Taylor expansion to produce linear approximations of non-linear functions. Mixtures of Gaussians can also be used to extend the Kalman filter to encompass situations in which multiple hypotheses are required, but each extension increases the complexity of the algorithm. This extensibility is one of the reasons for the popularity of the Kalman filter in state estimation problems. With some extension, the information filter is particularly suited to multi-robot systems where information from multiple sources must be integrated, but issues with complexity have resulted in the Kalman filter becoming the more popular of the two for the majority of problems [**?**].

3) Nonparametric Filters: In contrast to Gaussian filters, nonparametric filters discretise the probability distribution and do not require assumptions of linearity and Gaussian belief distributions. Instead, the state is approximated by a finite number of values which are taken from the belief state at any given time. The number of these values can be varied, and nonparametric filters converge to the correct state as the number tends towards infinity. Because this family of filters does not impose restrictions on the posterior density, they are useful for problems such as global localisation, which require the state to be represented in a complex form. Global localisation is the problem of determining the position of the robot without knowing its initial position, resulting in global uncertainty and the need for a multimodal belief distribution. Although as a result nonparametric filters are more computationally expensive than Gaussian filters, it is possible to vary the number of values used to represent the belief to suit the problem using *adaptive* techniques. Examples of these types of filters are the histogram and particle filters. The histogram filter decomposes a continuous space into some finite number of regions, each of which is assigned a probability based on the belief. Extensions include using dynamic decomposition techniques to use more coarse representations in regions with lower probability, and the use of *selective updating*, which updates only the parts of the space which are deemed important. Particle filters approximate the belief by drawing a number of hypotheses called *particles* randomly from the belief distribution. The most important part of the particle filter is the resampling step, which selects particles from the initial set proportional to an importance factor. This has the effect of concentrating particles in areas of high likelihood, reducing computational power spent in areas which are not relevant. Improving the sampling method and adapting the number of particles based on time and uncertainty lead to more effective and error-resistant particle filters. A property which is particularly useful to us is that particle filters are very easy to implement [**?**].

Algorithm 2 Basic Monte Carlo Localisation [**?**]

```
1:  Algorithm MCL($\mathcal{X}_{t-1}, u_t, z_t, map$)
2:  $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
3:  for $m = 1$ to $M$ do
4:      $x_t^{[m]} =$ sample_motion_model($u_t, x_{t-1}^{[m]}$)
5:      $w_t^{[m]} =$ sensor_model($z_t, x_t^{[m]}, map$)
6:      $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
7:  end for
8:  for $m = 1$ to $M$ do
9:      draw $i$ with probability $\propto w_t^{[m]}$
10:     add $x_t^{[i]}$ to $\mathcal{X}_t$
11: end for
12: return $\mathcal{X}_t$
```

4) The Markov Assumption: One of the reasons that these filters are so efficient stems from the assumption that the Markov property holds. While this may not actually be the case, the Bayes filter that forms the base of these filters is robust to the violation of some of the requirements of the property. The Markov property states that the future states of the system do not depend on past states. In other words, considering all previous states of the system provides no additional information for predicting future states; all prediction can be done using only the current state. Previous states do not have to be included in calculations, and as a result do not need to be stored, which leads to faster computation and lower storage space requirements.

5) Sensor & Motion Models: To actually use any of these filters to solve a localisation problem, it is necessary to model the sensors and motion of the robot. These models include some parameters representing the uncertainty attached to performing a specific action or receiving a certain measurement from a sensor. The motion model is used in the prediction step to determine the state transition probability $P(x_t \mid x_{t-1}, u_t)$; the probability of going from state x_{t-1} to x_t given that the action u_t was performed. The sensor model is used to calculate $P(z_t \mid x_t, map)$, which represents the likelihood of the measurement z_t being received in the state x_t on a given map. The model incorporates knowledge about the noise parameters of the operating environment and the sensor being used in the form of probability densities. For example, a range sensor might include densities for correct measurements, measurements of unexpected obstacles, sensor failures and random measurements [**?**].

Integrating the sensor and motion models into the particle filter results in an algorithm known as Monte Carlo localisation (MCL). The most basic version of MCL is shown in Algorithm **??**, in which $\mathcal{X}_{t-1}$ represents the particles from the previous time step, M the total number of particles, $x_t^{[m]}$ the state of the mth particle with the action u_t applied to it via the motion model, $w_t^{[m]}$ the importance weight of the mth particle, and $\bar{\mathcal{X}}_t$ the resampled set of particles. We use an adaptive implementation of MCL provided by the ROS system.

B. Route Planning

1) Probabilistic Roadmaps: A probabilistic roadmap (PRM) is a multi-query motion planning algorithm for robots in static workspaces [**?**] which solves the problem of determining a path between start and goal configurations while avoiding collisions with obstacles in the workspace. The roadmap algorithm consists of two phases; the learning phase and the query phase.

In the first step of the learning phase, the construction step, a graph G with vertices V and edges E is constructed. Free configurations are repeatedly sampled from the workspace and added to V. A free configuration is one where the robot does not intersect with walls or other obstacles in the workspace. Once a specified number of configurations have been sampled, the vertices in V are connected. Attempts to connect two vertices v and n are only made if they are within a neighbourhood specified by a maximum distance D, and a connection is only made if the path between the two nodes is also valid, i.e. there is no configuration on the path where

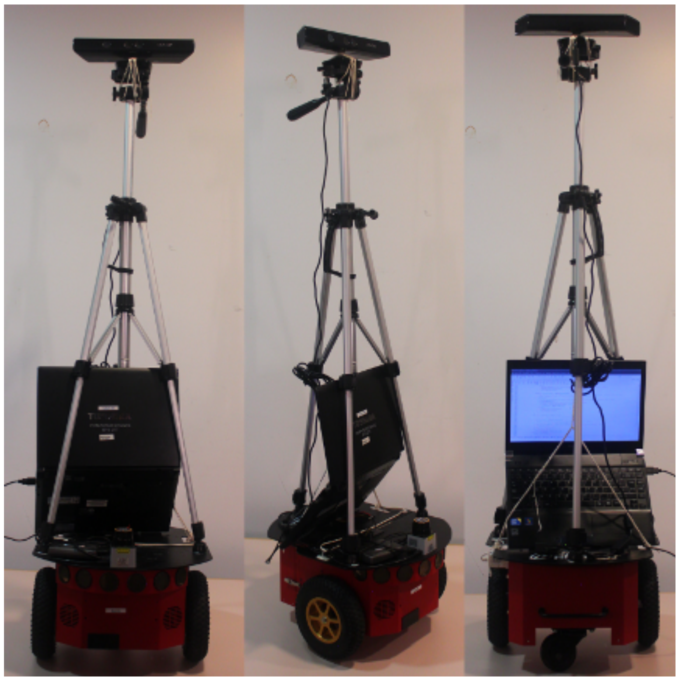

Fig. 1: The Pioneer 3-DX fitted with tripod mounted Kinect and Hokuyo laser rangefinder.

the robot intersects with walls or obstacles. If a connection is made, the edge from v to n is added to E. The second step of the learning phase is the expansion step, which is intended to improve the connectivity of the graph. This step can be skipped if a sampling algorithm which takes into account difficult (narrow) areas of the workspace is used in the construction step.

Once the graph has been constructed, the query phase is entered. In this phase, the roadmap is used to find paths between start and goal configurations s and g. In order to do this, s and g must first be connected to the graph using the same connection strategy used in the construction step. A graph search algorithm such as A* can then be used to find the shortest path between the two newly added vertices.

There are a number of ways in which the basic PRM algorithm can be improved. Most of these relate to the construction of the graph, as the graph is the part of the algorithm which most influences subsequent performance—if vertex placement is bad, then even if the graph search algorithm is good, it is possible that a path will not be found. The sampling method is particularly important, and many different techniques have been proposed to improve the coverage of the PRM. Roughly speaking, the techniques can be separated into uniform and advanced techniques. Uniform techniques include random sampling, sampling points on a grid, and splitting the workspace into cells and sampling some number of points from each of those subspaces. Uniform methods provide a simple but effective way of sampling, but may experience a drop in performance for more difficult workspaces, for example those with narrow corridors. In these cases, using advanced techniques such as medial axis sampling, which generates points at the midpoints between obstacles, can be effective [**?**]. The strategy used to connect vertices can also affect the structure of the graph, which in turn affects the query phase. Improvements to the query phase can be made by using different graph search algorithms, but unless the graph is extremely dense the query time is negligible. As an example, our implementation of Dijkstra's algorithm took 964 milliseconds to find a path between two points on opposite sides of a map with a roadmap populated with 5000 vertices. Path smoothing can also be used on the path found by the graph search algorithm to improve it further; our version is found in Algorithm **??**. In some cases, a path may not be found with the graph being used, and although expensive, re-generating the graph can sometimes remedy this problem.

2) *Rapidly Exploring Random Trees:* An RRT is a randomised data structure and algorithm that is also used to solve path planning problems. RRTs are designed to handle nonholonomic constraints and high degrees of freedom [**?**]. Instead of sampling points and then attempting to connect them as in PRMs, RRTs are biased towards moving into unexplored areas of the state space by sampling points and being "pulled" towards them [**?**]. There are several planners for RRTs, including a bidirectional planner where the tree grows from the start and goal vertices and uses an aggressive heuristic to connect the trees. In contrast to the multi-query PRM, an RRT is a single query algorithm; it is rerun every time a query is made. The properties of RRTs make them particularly suited to problems involving robot arms and robots with nonholonomic kinematics. As our platform can be considered holonomic, we mention RRTs for completeness.

C. *Exploration*

The problem of exploration revolves around taking actions to maximise information gain, given your current knowledge of the world. What is meant by information gain depends on the problem being solved. In many cases this is discovering the locations of walls and obstacles, but can also extend to more complex objectives such as determining the locations of specific objects, rooms, or people. The most basic form of exploration is the random walk, which ensures that any finite sequence of actions will be executed eventually [**?**]. "Eventually" is not good enough, however, and many improved exploration techniques have been proposed. Greedy techniques move the robot from its current location to the closest location that has not yet been explored [**?**]. Frontier-based exploration moves to boundaries between open space and uncharted territory in order to maximise its information gain [**?**]. Distributed approaches using coordinated multi-robot systems have become an active area of research, and novel techniques drawing on economics concepts have been proposed [**?**], [**?**]. As with many solutions using distributed computing, the use of multiple "processors" allows the task to be completed in a shorter time, but requires the implementation of algorithms that can make good use of these increased capabilities.

D. *Robot Vision*

III. Design

In this section we explain in detail the techniques, algorithms and hardware that we used to solve the task.

A. Hardware & Software

1) Robotics Platform: For the task, we were required to use a Pioneer 3-DX fitted with a Hokuyo URG-04LX laser rangefinder. The Pioneer is 455cm long, 381cm wide and 237cm high, and weighs 9kg, making it a relatively compact platform. It is fitted with six forward mounted and two side mounted sonar sensors. The robot is powered by three batteries which provide a run time of 8-10 hours, assuming no accessories are attached. The robot is equipped with a differential drive system which has a 0cm turn radius, making it essentially holonomic. The maximum translation and rotation speeds are 1.2m/s and 300°/s respectively [**?**].

The Hokuyo laser is mounted on the front of the robot. It has a scan area of $240°$ with a maximum radius of 4m and an angular resolution of $0.351°$. A scan is completed by the laser every 100ms. The laser is accurate to ± 20mm at distances between 20 and 1000mm, and $\pm 1\%$ of the measurement distance at 1000-4000mm [**?**].

The Microsoft Kinect is a video device which produces images much like a normal device, but has the additional ability to return a depth value for each pixel in the image. As such, the device is similar to a 3-D camera. The Kinect camera has a $57°$ horizontal and $43°$ vertical field of view with a nominal depth of 3.5m. Images from the camera have a resolution of 640x480 [**?**]. Interfacing the Kinect with the ROS framework is achieved by using the OpenNI SDK, an open source framework for 3D sensing development.

Figure **??** shows the assembled hardware.

2) Robot Operating System: The Robot Operating System (ROS) is an open source software framework developed by Willow Garage and the robotics community that provides libraries and tools for use in robot applications. The system comes with a series of *stacks*; user contributed packages which can be used to perform SLAM, localisation and various other tasks that the user may not want to implement themselves, which means that development can be focused on solving the problem at hand. ROS code can be written in a variety of languages, including C++, Python and Java, and can be run on a large number of platforms, with support for humanoid robots, arms, quadrotors and many mobile robots.

The system has a graph-like structure and makes use of data streams for communication between nodes, and as a result is robust to node failures. The use of data streams for communication allows for a high degree of modularity—a node written for one task can be used in the system for another task with little or no modification. Each node can have an arbitrary number of *publishers* for sending messages to *topics* and *subscribers* for receiving them. One of the major strengths of the system is that when a message is sent, it does not have to be sent to a specific node; once it is published, the message is available to any node which is subscribed to the topic on which it is published. A master node provides information about the state of the whole system, including node status and the currently active topics.

B. System Structure

Our implementation attempts to make the most of the ROS system, with separate nodes for each subsystem. The actions of the system are controlled by the control node which is based on a finite state machine. This node then hands off to other nodes when tasks need to be executed. The navigation node uses data from the localisation and PRM nodes to move between points in the map. Vision processing is done in a node written in C++ for speed, and the resulting data is used by the control node to make decisions about which actions to take.

1) Localisation Node: We independently developed two localisation nodes which implemented Monte Carlo localisation (MCL), but which were not sufficiently error free that we were confident in using them for the combined task. Instead, we made the decision to use the ROS implementation of AMCL which comes with the software installation. By using a tried and tested implementation, we reduce the likelihood of bugs causing problems with the localisation. This is very important, as to move safely through the space it is necessary to have an accurate position estimate. We had to tweak some parameters in the algorithm that we deemed necessary in the testing of our own algorithms while running in tandem with some of the other nodes that were to make up the final system. The AMCL defaults for filter updates are 0.2 metres for translation and 0.52 radians for rotation. We decreased both of these to 0.1 metres and 0.1 radians respectively in order to receive filter updates more often. A comparison of the tracking error of the three localisation algorithms can be found in Figure **??**.

Algorithm 3 Probabilistic Roadmap Generation

```
1: Algorithm generate_PRM($map$)
2: $V \leftarrow$ sample_vertices($map$)
3: for $v \in V$ do
4:     $a \leftarrow 0$
5:     while $a < A \wedge$ nconns($v$) $< C$ do
6:         $n \leftarrow$ get_closest($V, v$)
7:         if dist($v, n$) $< R_n$ then
8:             if $\neg$neighbour($v, n$) $\wedge$ vpath($map, v, n$) then
9:                 $E \leftarrow$ connect($v, n$)
10:            end if
11:        else if vpath($map, v, n$) then
12:            $E \leftarrow$ connect($v, n$)
13:        end if
14:        $a \leftarrow a + 1$
15:    end while
16: end for
```

2) PRM Node: The PRM node is used to find paths to navigate from one point on the map to another. Whenever a goal point is received by the node, it finds the shortest path from the robot's current position to the goal position by using Dijkstra's algorithm to search the road map generated when the node is initialised. We will now walk through Algorithm **??**, used for generating the roadmap. In line 2, points are sampled from the map using cell sampling. Cell sampling is a uniform sampling technique in which the map is split into a number of equally sized square *cells*, and a certain number of points are randomly sampled from within each cell. This sampling technique, although simple, provides very good coverage of the space, and is particularly good at sampling points in narrow corridors, as each cell is required to contain a specified number of points. We now iterate over the sampled points, hereafter called vertices, attempting to make connections between them. We connect the target vertex v to candidate vertices n in the

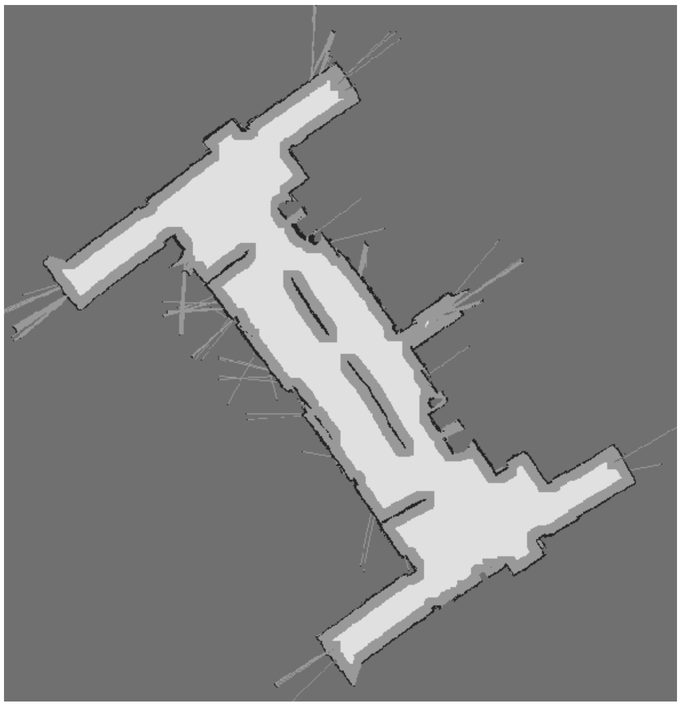

Fig. 2: Example of an inflated map used for PRM. Black pixels are obstacles. Grey pixels inside the map show the inflation applied to the obstacles.

graph in order of proximity to v. As some vertices may be unconnectable, we specify a maximum number of attempts A that should be made to connect a vertex to the graph. We also specify C, the maximum number of connections that a vertex can have. If the number of connection attempts a exceeds A, or the number of connections the target vertex has exceeds C, the algorithm will move on to the next vertex. When connecting vertices, the algorithm starts by selecting the vertex in the graph which is the shortest Euclidean distance from v. In line 7, a check is made to see if the candidate vertex is within the neighbourhood of the target vertex. Vertices within a Euclidean distance R_n of v are considered to be within its neighbourhood. If the candidate vertex is within the neighbourhood, v and n are connected only if they are not already connected by a path that is contained entirely inside the neighbourhood, and there is a straight line between the two vertices which does not pass through any obstacles. If the candidate vertex is not in the neighbourhood of the target, then a connection is made so long as there is a free path between the the two. These checks are performed in lines 8 and 11. The benefit of this is that vertices are not connected multiple times to other vertices in close proximity, but connect to vertices further away which means that shortcuts can be made between neighbourhoods. A vertex is guaranteed to be connected to the closest accessible vertex in its neighbourhood. When checking the validity of points and lines on the map, it is necessary to take into account the size of the robot. This can be done by checking a radius around each point for obstacles, but for the purposes of efficiency, we have taken a slightly different approach. Instead of checking the area around points, we increase the size of obstacles by the required safe distance from obstacles. In our case, this is slightly above half of the longest dimension of the robot. After applying this size increase, we receive an "inflated" map, an example of which can be seen in Figure **??**, which we use when generating the graph and for path planning.

When querying the PRM, a the robot's current position and the goal location are added to the graph and connected in the same way as other vertices. Dijkstra's algorithm is then run with the newly added vertices as parameters, and the shortest path between the vertices is returned. While using the raw path is viable, it can be further improved by smoothing. As seen in Algorithm **??**, redundant intermediate vertices are removed from the path in an iterative fashion. I is the number of smoothing passes executed, and P is a vector of waypoints along the path. P_n represents the nth waypoint in the path. The intermediate waypoint b is removed only if there is a valid path between waypoints a and c. The algorithm terminates either when the requested number of iterations has completed or if the path cannot be smoothed further. The benefit of smoothing is twofold; the path length becomes closer to the optimum, and the number of vertices in the path are reduced. The fewer vertices there are in the path, the less the robot must rotate during travel, and this results in a faster travel time from start to goal.

When a path has been found, a message is sent to a topic announcing that a path to the goal was found. In cases where a path cannot be found with the current graph that the PRM holds, the graph will be regenerated and searched again a number of times. If a path still cannot be found after the specified number of regenerations has occurred, the PRM will publish a message informing listening nodes of this fact so that they can handle the consequences.

Algorithm 4 Path Smoothing

```
1: Algorithm smooth_path($P, I, map$)
2: for $i = 0$ to $I$ do
3:     for $j = 0$ to $|P| - 2$ do
4:         $a \leftarrow P_i$
5:         $b \leftarrow P_{i+1}$
6:         $c \leftarrow P_{i+2}$
7:         if vpath($map, a, c$) then
8:             remove($P, b$)
9:         end if
10:    end for
11:    if path not modified then
12:        break
13:    end if
14: end for
15: return $P$
```

3) Navigation Node: The navigation node moves the robot along paths provided by the PRM. obstacle detection

4) Vision Node: For face detection capabilities, we decided to make use of camera data from the Kinect. Face detection in ROS necessitates the use of OpenCV or some other vision library. We decided to use OpenCV as it is written in part by the ROS developers, which should lead to easier integration. Communication between ROS and OpenCV requires the cv_bridge stack, which is not compatible with Java, only C++ and Python. We decided that C++ was a better candidate than Python, as face detection is very computationally expensive, and C++ is known to be a very fast language. The OpenNi

SDK publishes many ROS topics containing various data from the Kinect, of which, two are used that contain the depth and an image from the Kinect camera. These two topics are subscribed to by the C++ node that was created to allow the use of OpenCV with the Kinect image data. The output from OpenCV is then used in conjunction with the Kinect depth data against a model to determine whether or not a face has truly been detected. The C++ node publishes the location and area of the detected face along with the depth. This data is subscribed to by the control node which then uses the data accordingly.

talk about haar cascades here! A Haar cascade is a learned classifier which works with digital image features called Haar-like features. Haar cascades are usually trained on multiple sets of training data. The OpenCV package contains a range of cascade classifiers which can be used to perform a variety of detections. Some cascades are specialised to different forms of detection, for example the haarcascade_eye cascade would uniquely detect eyes on an image.

Algorithm 5 Face detection

```
1: Algorithm detect_face(image)
2: face_rects←haarDetect(image, haar_cascade)
3: for face in face_rects do
4:     if ¬validFace(face, depth) then
5:         remove(face_rects, face)
6:     end if
7: end for
8: publish face_rects
```

Algorithm **??** shows the face detection algorithm used. In line 1, a Haar feature detector is run on the image received from the Kinect video feed, from which a series of rectangles representing possible faces are received. Non-faces are removed from the array in line 4. If valid faces remain after the array has been checked, they are published to the network. Face validity is determined by Algorithm **??**. In line 2, depth values from a smaller rectangle of 20% width and height of the original rectangle, in the centre of the original rectangle are retrieved. We use the median depth of this smaller area as the actual depth of the potential face. If the median of the area exceeds a threshold of 3.5m as per the effective range of the face detection algorithm, the face is rejected. Otherwise, in lines 7–10, the expected depth and actual depth are compared. α is a tolerance applied to the value expected by the model, set to 15%. If the actual depth is within this tolerance range, the face is considered valid.

As we had access to depth data from the Kinect, when a person was detected we could determine their approximate position and send this point to the PRM to move to them, rather than exploring in the general direction in which the person was detected. This has the additional benefit that if a person is detected over a low obstacle, the PRM will be aware of it and be able to navigate around it.

5) Control Node:

IV. Experimentation

In this section we display the results of experiments carried out to determine the best algorithms, parameters or strategies to use in various parts of our system.

Algorithm 6 Face validity check

```
1: Algorithm validFace(face_rect, depth)
2: depth_pixels←getCentreRegion(face_rect)
3: median←getMedian(depth_pixels)
4: if median>threshold then
5:     return False
6: else
7:     expected←expectedDepth(face_rect)
8:     tolerance← $\alpha$·expected
9:     diff← |median − expected|
10:    if diff>tolerance then
11:        return False
12:    end if
13: end if
14: return True
```

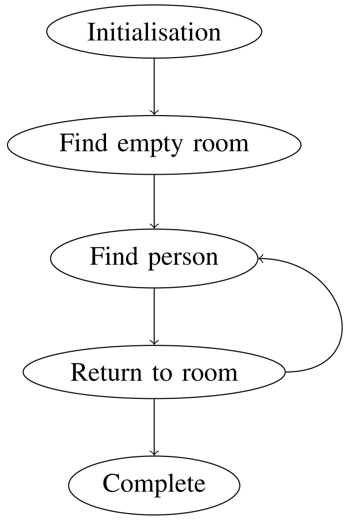

Fig. 3: Simplified version of finite state machine used in the control node.

A. MCL Tracking Error

1) Aim: The aim of this experiment is to compare the tracking error of three different MCL implementations; the built in ROS AMCL and two of our own implementations.

2) Method: As each MCL implementation has many different variables, we will use the set of variables that resulted in the best performance in past experiments. To provide accurate results, we require a ground truth, which will be obtained from the Stage simulator. We will run the experiment three times for each implementation and record the tracking error every second while the robot moves, until it reaches the end of its path. The tracking error is defined as the Euclidean distance between the ground truth and the estimated pose provided by the localisation at each point at which a measurement is taken. The robot will start from the same initial position and follow the same path, seen in Figure **??**, in each experiment.

3) Results: Figure **??** shows the tracking errors for each implementation, with error bars representing the standard deviation over three runs. Apart from two anomalous points at 13 and 25 seconds at which all implementations experience

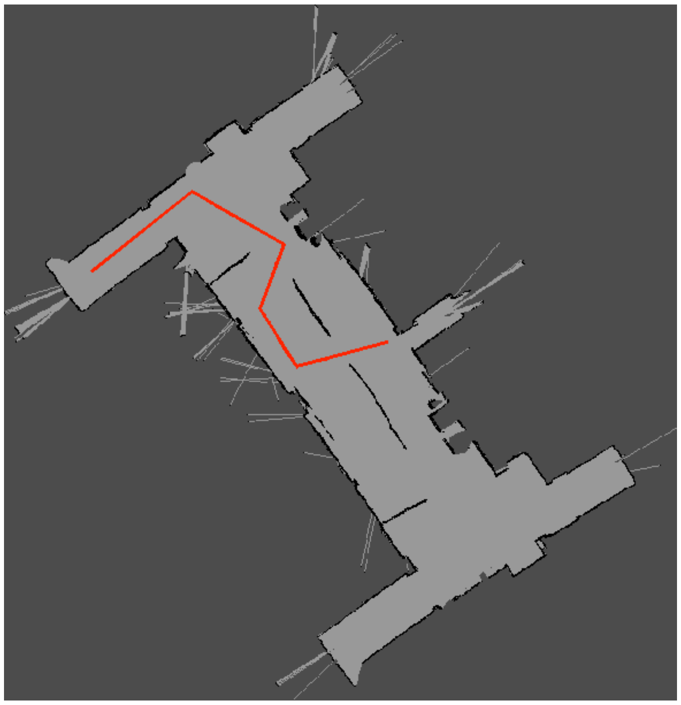

Fig. 4: Path used for tracking error experiment.

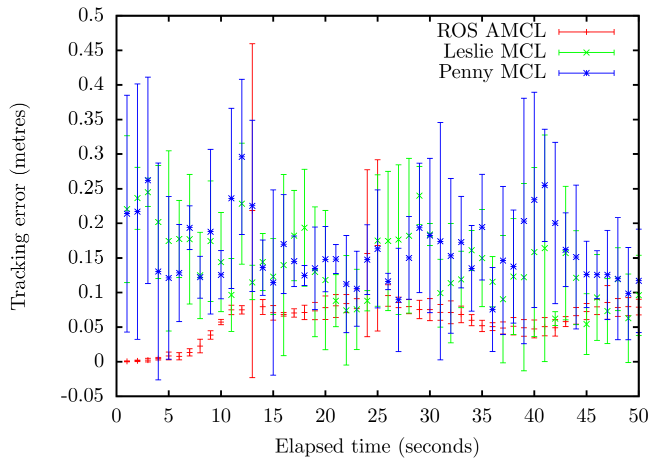

Fig. 5: Comparison of Tracking error of ROS AMCL and two independently developed MCL algorithms

an increase in tracking error, the deviation of AMCL is significantly lower than the other two implementations. The average deviation over the three runs for AMCL is 0.019m, compared to averages of 0.071m and 0.085m for Penny and Leslie respectively. It is clear that to obtain accurate position estimates we should make use of ROS AMCL.

B. PRM connection strategies

1) Aim:

2) Method:

3) Results:

C. PRM sampling methods

1) Aim:

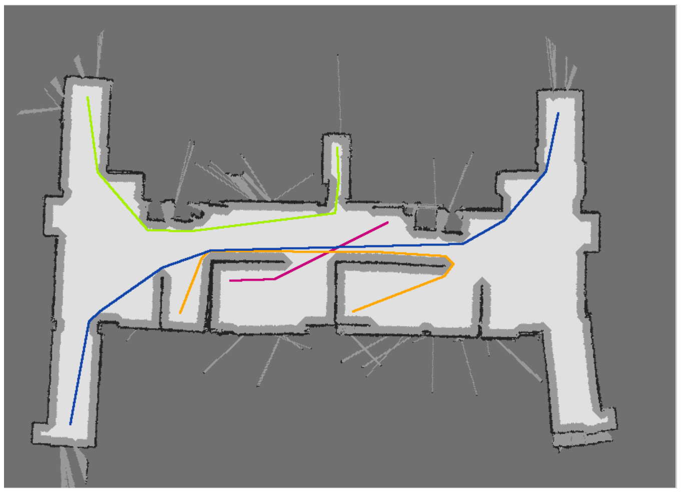

Fig. 6: Optimum paths used for comparison of connection strategies and sampling methods.

2) Method:

3) Results:

D. Haar cascade efficacy

1) Aim: The aim of this experiment is to find the Haar cascade most suited to perform face detection with the Kinect. We wish to find the cascade which has the highest specificity and sensitivity; all actual faces should be classified as faces, and anything which is not a face should be ignored.

2) Method: To test the different Haar cascades we captured a set of images using the Kinect camera which all included faces in different surroundings and at different angles. We then processed each image using our cascade-based face detection node which works on still images, using the frontalface_default, frontalface_alt, frontalface_alt2 and frontalface_alt_tree cascades. We manually analysed the processed images, which contain rectangles indicating the areas detected as faces. In order to generate the data, we looked at the image output from the program for each cascade and analysed the number of faces that were correctly detected (true positives), the number of non faces not detected (true negatives), the number of falsely detected faces (false positives) and finally, the number of faces which were completely missed (false negatives). Once we had all these values for each image, we averaged the sensitivity for each Haar cascade to find their final value.

3) Results: From our results of the above experimentation it was obvious all cascades did not perform well on profile faces (i.e faces at a near 90 degree angle). This was expected as the cascades we used were specifically trained on frontal face images and therefore the detection was failing. As soon as enough information could be gathered from the image regarding the face, the default and alt cascades both detected the face and performed consistently. However, the other two detectors required the face to be at less of an angle in order to be detected. On the images which had different surroundings we found that again the alt and default cascades performed rather well. As is obvious in the ROC graph in Figure **??** the best performer was the alt cascade. Although its sensitivity value of 0.744 was a little lower than the default value of 0.758,

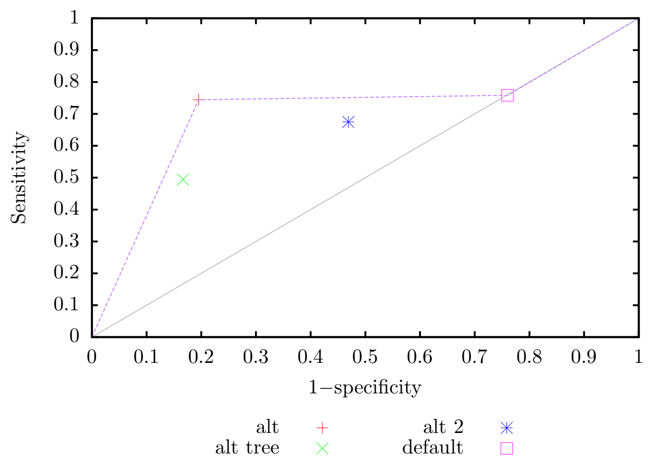

Fig. 7: ROC graph of frontalface haar cascade results.

the alt cascade performed a lot more effectively detecting true negatives (i.e. correctly identifying no faces exist). This result pushed the 1-specificity reading for the alt cascade towards the left, highlighting it as the best candidate amongst all cascades tested. Although the default cascade performed better in the sensitivity reading, it also performed worst in the 1-specificity reading, showing it wrongly classified non faces as faces frequently.

4) Conclusion: In conclusion, from the data gathered from the experiment we can see that the Haar cascade which performed above other cascades in detecting faces was the frontalface_alt. We saw that this cascade, with a good degree of consistency, classified actual faces as faces (true positives). As we decided to use the Kinect as a camera, we were aware it is possible to extract depth information for the object in view. Given that this cascade still detected false positives we decided to use the depth information to identify false positives, by using knowledge of the size of human faces. To do so, we developed a model which helps prune non-face objects which are picked up by the cascade. This would allow us to improve the performance of this cascade by reducing the number of false positives.

E. Kinect depth model

1) Aim:

2) Method:

3) Results:

F. Exploration coverage

1) Aim:

2) Method: The robot's maximum movement speed and rotation speed were set to 0.7m/s and 0.4rad/sec respectively. The amount of the map that can be explored in a fixed amount of time varies greatly depending on the movement speed of the robot. The maximum distance of the FOV was set to 3.5m, and the minimum to 0.3, the maximum and minimum effective ranges of our face detector.

3) Results:

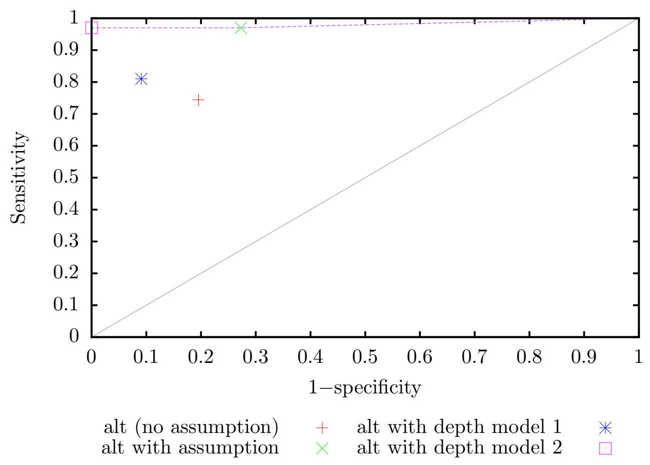

Fig. 8: ROC graph of frontal_face_alt cascade with the assumption of subjects facing the camera and depth models applied.

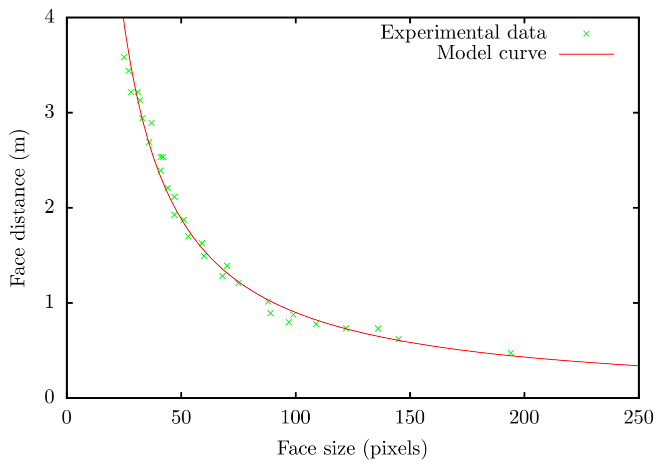

Fig. 9: Depth model for the centre of the Kinect's field of view.

G. Meeting room availability confirmation

1) Aim:

2) Method:

3) Results: medium failures: 2 times, once found one person, other time didn't find anyone. both failures due to AMCL issues. Short failures 4, due to AMCL. Long failures, 3 AMCL failures.

H. Gathering meeting attendees

1) Aim:

Strategy	Sampling method	
	Cell	Grid
2m→1m	91.3	**89.4**
4m→2m→1m	**94.0**	86.4
6m→4m→2m	93.5	83.5
8m→6m→4m	93.7	82.8

Fig. 10: Percentage coverage after eight minutes of exploration with different strategies and sampling methods.

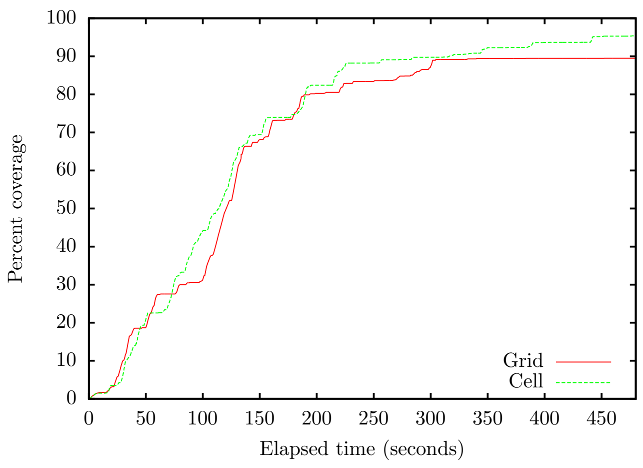

Fig. 11: Comparison of the best grid and cell based exploration strategies.

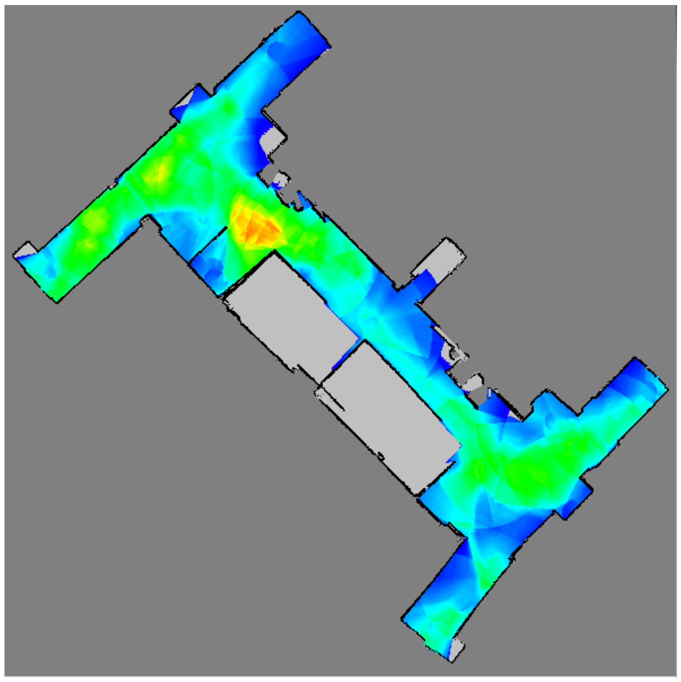

Fig. 12: Heat map showing coverage over an 8 minute run using the best cell based strategy. The colour gradient goes from red (high) to blue (low)

2) Method:

3) Results:

V. Discussion

A. Performance

B. Potential Improvements

the path generated to do exploration could be improved by using heuristic techniques. multiple robots

C. Conclusions