Occupancy Grid Maps for Localization and Mapping

Adam Milstein
University of Waterloo
Canada

1. Introduction

In order to perform motion planning it is usually necessary to define some representation of the environment and have some method of determining the robot's location in that environment. Mapping is the problem of determining the representation, while localization is the problem of finding the robot's position. Many probabilistic techniques for localization depend on the map being defined as a finite sized set of landmarks which the robot's sensors observe, giving their relative displacement from the robot. However, physical sensors do not usually detect landmarks unambiguously. Instead, they report the distance to the nearest obstacle, or return an image of the environment. In order to use a landmark based algorithm, the sensor readings must be pre-processed in a separate step to convert the raw sensor data into a set of detected landmarks, such as in [Leonard and Durrant-Whyte 1991]. The additional step introduces more error into any algorithm, as well as discarding much of the sensor information which does not detect any landmark.
One of the primary drawbacks of landmark based maps is the data association problem. Because raw sensor data is not labelled with the correct landmark detected, the sensor processing must somehow determine exactly which landmark was observed. If mistakes are made the localization and mapping algorithms which depend on the sensor data will fail. In order to compensate for the data association problem, many localization and SLAM algorithms include a method for determining the associations between the sensor data and the landmarks, however these techniques add significantly to the complexity of the solutions. Also, they do not solve the problem of actually finding landmarks in the raw sensor readings. Some examples of these algorithms include GraphSLAM [Folkesson and Christensen 2004] and Sparse Extended Information Filters (SEIF) [Thrun et al. 2004], both of which can be implemented to handle data associations in a probabilistic way as described in [Thrun et al. 2005]. Even with these integrated solutions, the data association problem adds a significant amount of error.

2. Occupancy Grid Maps

One common technique for map representation that does not suffer from data associations is to use occupancy grid maps to approximate the environment. An occupancy grid map represents the environment as a block of cells, each one either occupied, so that the robot cannot pass through it, or unoccupied, so that the robot can traverse it. Unless your

environment is composed entirely of cubes, occupancy grid maps cannot be absolutely accurate, but by choosing a small enough cell size they can provide all the necessary data. Any sensor will report the status of a set of grid cells that can be checked without reference to the rest of the map. An early implementation of occupancy grid maps was used in [Moravec 1988] to automatically generate a map of the environment. Sensor readings are compared to the map, altering the probability that observed cells are occupied. For example, a sonar sensor returns the closest object within a cone, so the cells in the volume of the cone closer than the reading are probably unoccupied. Moravec represents each cell as a probability of being traversable and initializes them to an unknown value. He describes a probabilistic technique to update cells for various types of sensors and gives a technique to allow the map to be updated as the robot moves. Unfortunately, this technique is not actually localization and does not help the robot know its own position. The map is maintained relative to the robot, rather than in a global frame of reference. In other words, the robot is assumed to be at a fixed location, while the map moves around it. As the robot moves, the map is blurred according to the motion. The robot's sensors can correct the map in its immediate area, but unobserved portions of the map must blur into uselessness. There is also no way to discover the robot's location in reference to previously visited locations.
Although the technique is problematic as a localization algorithm, it provides a very powerful way to represent the environment. Using an occupancy grid map allows the raw sensor data to be used without trying to detect and identify landmarks. Also, since raw data is used, no information is discarded because it does not correspond to a landmark. The only problem is that there are a huge number of map features, one for each grid cell. Algorithms which consider the relation of the robot to a set of distinct landmarks cannot be applied when the number of features is so large. Thus, using occupancy grid maps limits the type of localization and mapping techniques that can be used.

2.1 Mapping Technique

To create an occupancy grid map it is necessary to determine the occupancy probability of each cell. In order to do this efficiently the assumption is often made that map cells are independent. Although this is not strictly accurate, especially when considering adjacent cells representing the same physical object, it greatly simplifies the mapping algorithm without significantly increasing the error. As a result, the probability of a particular map, m, can be factored into the product of the individual probabilities of its cells. The map depends on the robot's location history, $x^t = \{x_0, \ldots, x_t\}$, and the sensor readings at each location in the path, z^t

$$p(m \mid x^t, z^t) = \prod_n p(m_n \mid x^t, z^t) \tag{1}$$

The probability of a particular cell is easy to determine given the robot's position and sensor readings, since it is determined by whether the robot observes the cell as unoccupied or occupied. Since the probability is determined by the robot's entire history, all these sensor readings must be taken into account. The mapping algorithm usually builds the cell probabilities up iteratively, considering each $\{x_t, z_t\}$ from time t = 0 to the most recent reading. Although these readings could be considered in any order, the iterative processing makes the most sense, allowing additional readings to be added and leading eventually to simultaneous mapping and localization (SLAM) solutions such as described in section 4.

With occupancy grid maps, the mapping step must determine the probability of each cell, as represented by equation (1). Proceeding iteratively, the map cells are updated according to the position and sensor readings. Of course, it would require significant processing to update the entire map on each step, but this is unnecessary. Only the cells which are actually observed need to be updated. Each cell that is perceived by the sensor given the robot's position is updated depending on whether the sensor indicates it is occupied or unoccupied. Although the map is defined by the occupancy probability, for simplicity the actual values for each cell, $p(m_n \mid x^t, z^t)$, are calculated in log odds form.

$$p(m_n \mid x^t, z^t) = 1 - \frac{1}{1 + e^{l_{t,n}}} \tag{2}$$

$$l_{t,n} = l_{t-1,n} + \log \frac{p(m_n \mid x_t, z_t)}{1 - p(m_n \mid x_t, z_t)} - \log \frac{p(m_n)}{1 - p(m_n)} \tag{3}$$

$p(m_n)$ is a constant prior occupancy probability, so the only important part of equation (3) is $p(m_n \mid x_t, z_t)$, which is called the inverse sensor model. Although a highly accurate inverse sensor model is difficult to determine, a simplified implementation that returns a high value if the sensors report an object in the cell and a low value otherwise is often acceptable. Occupancy grid mapping updates a map according to a sensor reading at a location so that, as evidence accumulates, the map becomes correct. Of course, the success of the mapping algorithm depends on the location x_t being correct, just as the success of localization depends on an accurate map.

3. MCL

Monte Carlo Localization uses models of various sensors, together with a recursive Bayes filter, to generate the belief state of a robot's location. In fact, MCL is a specific instance of a POMDP (Partially Observable Markov Decision Process). A standard form of MCL uses a motion model to predict the robot's motion together with a sensor model to evaluate the probability of a sensor reading in a particular location. The sensor model necessarily includes a static map of the environment. The algorithm can be applied to virtually any robot with any sensor system, as long as these two models can be created. One common implementation where MCL is very successful is on a wheeled robot using a range sensor such as a laser rangefinder. A benefit of this combination is that the map and location used by the algorithm are in a human readable format. Although I give the general algorithm in the following sections, which should be applicable to other robots, where application specific details are required, I assume the type of robot as described.

Other localization algorithms than MCL exist, but they are currently much more limited than MCL and require specific environment features in order to be effective. Most other localization algorithms require that the map be composed of discrete landmarks and often they increase in runtime with the size of the map. Extended Kalman Filter (EKF) localization [Leonard and Durrant-Whyte 1991] is an alternative technique that has both of these problems, which are exacerbated when landmarks cannot be identified exactly. Even with various optimizations to improve execution, such as using the unscented transform (UKF localization) [Julier and Uhlmann 1997] or multi hypothesis tracking (MHT), the algorithm still applies primarily to feature based maps [Thrun et al. 2005]. Since a large, indoor environment is unlikely to have discrete, unambiguous features, these techniques are ineffective for the type of problem we are considering. In order to apply them it is usually necessary to preprocess the map as in [Leonard and Durrant-Whyte 1991] to create an artificial landmark based map. The map processing step introduces additional error and discards much of the information provided by the original map. Another serious problem is that these alternative techniques cannot handle multiple hypotheses of the robot's location. Each one maintains only a single Gaussian representation of position. MCL, in contrast, can maintain multiple separate locations. Markov localization using probabilities over a grid is also possible, however it increases in runtime with the size of the state space. Since a robot in a real, dynamic environment requires a real valued state space, true Markov grid based localization is usually impossible. Because of these drawbacks, MCL is currently one of the most effective localization techniques and the most commonly used, especially for real robots operating in real environments.

3.1 Recursive Bayes Filter

MCL is an implementation of a recursive Bayes filter. The posterior distribution of robot poses as conditioned by the sensor data is estimated as the robot's belief state. A key detail of the algorithm is the Markovian assumption that the past and future are conditionally independent given the present. For a robot, this means that if its current location is known, the future locations do not depend on where the robot has been. In virtually any environment this is the case, so making the assumption is reasonable in general.
To produce a recursive Bayes filter, we represent the belief state of the robot as the probability of the robot's location conditioned by the sensor data, where sensors include odometry (u_t).

$$Bel(x_t) = p(x_t \mid z_t, z_{t-1} \ldots, z_0, u_t, u_{t-1}, \ldots, u_0) \tag{4}$$

x_t represents the robot's position at time t, z_t the robot's sensor readings at time t and u_t is the motion data at time t. To simplify the subsequent equations we use the notation $a^t = \{a_t, \ldots, a_0\}$.

$$Bel(x_t) = p(x_t \mid z^t, u^t) \tag{5}$$

While this equation is a good representation of the problem, it is not much use since it cannot be calculated as is. By applying a series of probabilistic rules, together with the Markovian assumption, equation (5) is converted into a more usable form.

$$Bel(x_t) = \eta p(z_t \mid x_t) \int_{x_{t-1}} p(x_t \mid u_t, x_{t-1}) p(x_{t-1} \mid u^{t-1}, z^{t-1}) dx_{t-1} \tag{6}$$

Obviously, $p(x_{t-1} \mid z^{t-1}, u^{t-1})$ is $Bel(x_{t-1})$, giving us the recursive equation necessary for a recursive Bayes filter. η is a normalization constant that can be calculated by normalizing over the state space. $p(z_t \mid x_t)$ is the sensor model, representing the probability of receiving a particular sensor reading given a robot's location. Finally, $p(x_t \mid x_{t-1}, u_t)$ is the motion model. It is the probability that the robot arrives at location x_t given that it started at location x_{t-1} and performed action u_t. The sensor and motion model are representations of the physical components of the robot and must be determined experimentally for each robot and sensor device.

3.2 Particle Approximation

It would appear that, given the two models, equation (6) is all that is necessary to perform localization with MCL. Unfortunately, a problem occurs with the integral. The equation requires integrating over the entire state space. Although we can evaluate the models at any point in the space, there is no closed form to the integral. Further, even a simple robot moves in a continuous, 3 dimensional state space with an x and y location together with an angle of rotation. Calculating the integral over this space is impossible, especially for a real time algorithm. In order to solve this problem, we approximate the continuous space with a finite number of weighted samples.

$$Bel(x_t) \approx \left\{x_t^i, w_t^i\right\}_{i=1,\ldots,N} \tag{7}$$

The integral over the space becomes a sum over the finite number of particles.

$$Bel(x_t) = \eta p(z_t \mid x_t) \sum_N p(x_t \mid u_t, x_{t-1}) p(x_{t-1} \mid u^{t-1}, z^{t-1}) \tag{8}$$

Of course, approximating the space results in a certain amount of error when low probability locations are not represented. If the robot is really at one of these locations it can never be localized. However, if the number of particles is well chosen MCL works properly in most situations.

3.3 Resampling

One problem with using a finite set of particles to represent an infinite space is that the weight of particles representing a low probability location will quickly decrease and is unlikely to ever increase again. Similarly, if there are too few particles representing a high probability location, they will disperse and eventually lose the robot's position. What is needed is a method for relocating low probability particles to high probability locations and recalculating their probability. The method used in MCL is resampling. After the particles are weighted by the sensor model they are resampled to represent the high probability locations. N particles are chosen randomly from the list of N weighted particles, with probability according to their weight. These particles are chosen with replacement, so that after a particle is chosen it remains in the original list and has the same probability of being sampled again. A high probability particle might be selected several times and so multiple copies might occur in the new list, while a low probability particle might never be chosen at all and its location would die out. The resampled list will thus have multiple particles in high probability locations and none in low probability ones. Another effect of resampling is to set all the sample weights to 1 / N. Instead of having individual weights representing the probability of a location, the number of particles indicates the probability. A high probability location will have many particles and thus, if the robot is present, it is likely to be tracked as it moves. Of course, low probability locations will die out and be unrepresented, so localization will fail if the robot is truly at one of these positions.

3.4 Bias

Representing an infinite space with a finite number of samples necessarily introduces some error. In order to accurately represent high probability locations, the particle filter discards lower probability regions as their low likelihood particles are not selected during the resampling process. Bias is the name given to the problem that MCL tends to consider only the highest probability locations, letting others be removed. The effect is that MCL is biased towards areas that have a large number of particles, tending to converge, over time, to a single cluster in the highest probability location. However, in most situations the high probability location contains the robot and so the convergence provides the correct result.

3.5 Algorithm

As the robot moves, it reports its odometry and sensor data to the MCL algorithm. After each reported move every particle is moved according to the random motion model, based on the motion actually reported. The particles are then updated with a weight determined by the sensor model for the particle's location. Finally, the particles are resampled by repeatedly choosing samples randomly, with replacement, from the current set, according to the weights assigned by the sensor model.

```
1:    Create a set {$x_t^{[n]}$, $w_t^{[n]}$} from $X_{t-1}$ by repeating N times:
1.1:          Choose a particle $x_{t-1}^{[n]}$ from $X_{t-1}$. Because of the resampling step this
              particle may be selected either iteratively or randomly.
1.2:          Next, draw a particle $x_t^{[n]} \sim p(x_t \mid u_t, x_{t-1}^{[n]})$. This particle is the result of a
              random motion according to the motion model.
1.3:          Set the weight of the particle using the sensor model: $w_t^{[n]} = p(z_t \mid x_t^{[n]})$.
2:    Resample randomly according to weight from {$x_t^{[n]}$, $w_t^{[n]}$} into $X_t$, which causes the
      particle weights to become uniform.
```

Table 1. MCL Algorithm

The effect of resampling is to replace the weight of the individual particles with the number of particles at that location. On the robot's next move the particles at a high probability location will spread out as they are moved randomly according to the motion model, with at least one landing in the robot's new location. Then the resampling will cause more particles to appear at the correct location, while incorrect locations die out. Assuming that the models and map are accurate, MCL will correctly track the robot's changing location. Various parameters can be tuned manually to adjust the rate of convergence and the behaviour of the models. Once the belief over the robot's location is generated, a single location for the robot can be found by looking at the mean of the particles.

3.6 Sensor Model

Corrections to the robot's location as determined by dead reckoning are made according to the robot's other sensors. The sensors, usually some type of rangefinder device, determine the weight of each particle. The weight is calculated according to $p(z_t \mid x_t)$ which represents the sensor model, the probability of getting a particular sensor reading given a suggested robot location. The sensor model depends heavily on the exact physical sensors installed on the robot, so there can be no general equation. Since the model is not sampled as is the motion model, it is often implemented as a large, precalculated table, where any particular sensor probability can be quickly looked up. A table implementation allows a more complex function to be used than could be calculated in real time. One function that is sometimes used for a laser rangefinder device gives the probability of each possible returned range value, given each possible actual distance to a wall. Such a function can be composed of a Gaussian distribution centered on the actual wall distance, since that distance is the most probable return value, together with other functions depending on the features of the physical device. Common additions are an exponential function multiplied by a linear one representing false negative values and another exponential function representing false positives. The overall probability of a laser reading, which is composed of multiple range values in different directions, is the product of the probability of each range value. Given a robot position, the distance to the wall along each sensor ray can be determined from the map and the probability of the range value returned given that distance can be looked up from the table. These probabilities are then multiplied together to get $p(z_t \mid x_t)$.

3.7 Motion Model

The motion model $p(x_t \mid x_{t-1}, u_t)$ is a critical part of MCL. Unlike the sensor model, which gives the probability of getting a specific sensor reading at a particular location, it is necessary to sample from the motion model. Given a starting location and a reported motion (x_{t-1} and u_t), MCL requires that we be able to choose a final location randomly according to the motion model. This requirement precludes us from using any motion model that is very complex. In fact, most motion models are a combination of simple Gaussian distributions. For a holonomic wheeled robot, the most common representation is with two kinds of motion leading to three kinds of error. Each movement of the robot is represented as a linear movement followed by a stationary turn. Although a particular robot probably does not follow these exact motions, if we break the robot's motion into small increments we can use them as an approximation.

These two motions are often implemented using two Normal distributions for many common robots. However, the algorithms described in this section should work for any model, provided it is possible to sample from it. In general, some collection of Gaussians works well, since they are often good approximations to a physical system while at the same time being easy to sample from and optimize.

3.8 Raytracing

Calculation of the sensor model $p(z_t \mid x_t)$ involves determining the probability of receiving a particular sensor reading given the location in the environment. For a laser rangefinder the readings are distance measurements. Given a robot's possible location in the map, the expected distance to the wall is usually determined by raytracing from the robot to the nearest wall. The robot's physical sensors determine its actual distance to the wall and, once the distance from the map and the distance from the world are known, the probability can be calculated either mathematically or by a table lookup.

4. FastSLAM

4.1 SLAM Problem

The problem of determining the map and robot position at the same time is called Simultaneous Localization and Mapping, (SLAM). It involves finding the distribution over a state space which includes both robot position and the complete map. The given data is the sensor and odometry information from the start until the current time. Even the definition of SLAM results in two different problems. Determining the map and location during operation of the robot requires finding only the current location x_t as well as the static map m. That results in the problem of online SLAM, which is intended to localize the robot during operation while also creating the map. Online SLAM is concerned with determining $p(x_t, m \mid z^t, u^t)$, which is the state at the current time. Using new information to update old estimates is not part of online SLAM, even though new information could update the map so that past localization could be corrected. The other SLAM problem is called the full SLAM problem, and it involves finding the complete pose history of the robot and the map. The probabilistic formula is $p(x^t, m \mid z^t, u^t)$, since we want all of the robot's states, instead of just the current one. The difference is that full SLAM uses current data to correct past estimates. Online SLAM is used to localize the robot dynamically while full SLAM is often an offline algorithm concerned with finding out what the robot has already done. If the robot needs to make decisions based on its location, then it is necessary to use online SLAM. If the objective is to determine where a robot controlled by some other method, for example a human driver, has been, then full SLAM is more powerful. Both the problems have an application and the various solutions are similar, although not identical. In fact, online SLAM is the result of removing the past poses from the full problem using integration.

Although SLAM is technically the definition of a particular problem, it is also the name given to the current set of solutions to the problem. These solutions all have several common elements which are shared by all effective solutions to both the online and full problems. One of the most important factors of the SLAM solutions is correspondences. Since SLAM considers the map as well as the robot pose, there must be some definition of a correct map. In SLAM, maps are defined as sets of objects and a correct map is one that has each object in the correct location. Of course, sensors do not report the location of specific objects, so it is necessary to find which object each sensor reading corresponds to. Unfortunately, it may be difficult to determine exactly which object is being observed. As we have seen, heuristic methods can be used to filter the raw sensor data into object locations, but any such technique will have a certain percentage of errors. Some SLAM algorithms explicitly take correspondence probabilities into account, adding yet another term to the posteriors. If we define c_t to be the set of correspondences between sensor readings and objects at time t, the online SLAM problem becomes $p(x_t, m, c_t \mid z^t, u^t)$, while the full problem is $p(x^t, m, c^t \mid z^t, u^t)$. Of course, increasing the size of the state space significantly increases the complexity of the problem and thus the run time of the solution. Many SLAM algorithms can be proved to eventually converge to the correct map, but only if the objects can be identified correctly. If identification is not certain, the guarantee of convergence is lost. The problem of determining which object is detected by a sensor reading is called the data association problem and it is the central drawback to SLAM algorithms. In effect, SLAM usually creates a landmark based map, rather than a pure occupancy grid map. As we have seen, correctly identifying landmarks in localization is a difficult problem, which can be overcome by using raw sensor readings in the MCL algorithm. However, the corresponding SLAM solution suffers from additional problems.

4.2 FastSLAM Derivation

Simultaneous Localization and Mapping is divided into two slightly different domains. The first, called online SLAM, is the problem of finding the robot's current pose x_t and the map m, given the sensor readings z^t and odometry u^t. The more complex problem is to find the robot's path $x^t = \{x_1, \ldots, x_t\}$ given the same data. Finding the complete path is called the full SLAM problem. Obviously, full SLAM is the more complete problem but online SLAM can be derived from full by integrating out the past poses, as shown in equation (10).

$$\text{Full SLAM}: \; p(x^t, m \mid z^t, u^t) \tag{9}$$

$$\text{Online SLAM}: \; p(x_t, m \mid z^t, u^t) = \int_{x_{t-1}} \int_{x_{t-2}} \cdots \int_{x_1} p(x^t, m \mid z^t, u^t) dx_1 dx_2 \ldots dx_{t-1} \tag{10}$$

The reduced problem is called online SLAM because it is simplified enough to be solved in real time, whereas most full SLAM solutions require offline processing.
One of the benefits of FastSLAM [Montemerlo et al. 2002] is that it simultaneously solves both the online and full SLAM problems. Of course, any solution to the full SLAM problem also produces a solution to online SLAM, but FastSLAM, although it works in real time, solves for the entire path of the robot. The key to FastSLAM is that if the robot's location is known, the locations of objects in the environment are independent. Thus, the SLAM problem can be factored into localization and mapping problems.

$$p(x^t, m \mid z^t, u^t) = p(x^t \mid z^t, u^t) \prod_n p(m_n \mid x^t, z^t) \tag{11}$$

The first term is obviously a localization problem which requires finding the robot's path x^t given its odometry and sensor data. The second term is the mapping problem which finds a particular feature's position, m_n, given the robot's path and sensor readings. Equation (11) leads to an iterative algorithm for FastSLAM where the robot's position is calculated, and then the map is updated based on that position. Unfortunately, it is unlikely that at every step a single location for the robot can be derived. Nor can the algorithm rely on a probabilistic position, since the factoring is only valid given a fixed x^t. These requirements lend themselves to a technique that represents the robot's location as a finite collection of exact states. Fortunately, such a technique, Monte Carlo Localization (MCL), exists.

4.3 Occupancy Grid FastSLAM

Unlike most SLAM algorithms, it is possible to use FastSLAM on an occupancy grid map directly, without first processing it for landmarks. Since the algorithm updates the map with reference to a given robot position, the specific representation of the map as Gaussian landmarks is not required. Instead, an occupancy grid can be used and updated according to the standard techniques. As in [Moravec 1988], observed cells are updated according to whether the sensors observed them as occupied or unoccupied. However, since each particle represents an exact robot path, there is no need to blur past readings. The grid cell implementation even overcomes the data association problem, since landmarks can no longer move, the cell being observed is exactly determined by the robot's location. An implementation called DP-SLAM, [Eliazar and Parr 2004] was able to successfully localize and map a real environment including a large loop.

The only serious problem with FastSLAM occurs with the difficult situation of loop closure. In other algorithms, when the robot re-enters known territory it becomes necessary to search a much larger set of landmarks for correspondences, possibly the entire set. However, FastSLAM represents all possible robot positions in a finite set of samples. When it closes a loop, it can only be successful if some particle has followed the true path. The longer the loop, the greater the uncertainty of the robot's position. As uncertainty increases, the number of particles necessary to represent the belief also increases. Eventually, there will not be enough particles to represent the distribution and the correct location may be lost. FastSLAM alone suffers the problem, since all other SLAM solutions use the correlations to determine the position. The problem with particle filters is that they only represent the highest probability region of a distribution, whereas the Gaussian distributions used by other techniques represent the entire distribution. Of course, particles can represent much more complex and nonlinear distributions than is possible with Gaussians. The drawback to using particle filters in FastSLAM is that the number of particles maintains the diversity of the robot's position, and as soon as the uncertainty goes beyond the number of particles, the algorithm may fail. Thus, the size of the particle set must be tuned to the environment, based on the size of the longest loop, and increasing the number of particles to this extent may make the algorithm inefficient.

Grid based FastSLAM is performed by combining the MCL particle filtering with the occupancy grid mapping algorithm using Rao-Blackwellized particle filters [Montemerlo et al. 2002; Thrun et al. 2005]. Each particle consists of both the robot's state and an occupancy grid map. Of course, particle filtering could be used over the entire state space. However, this would require a number of particles exponential in the number of state variables, in this case the number of cells in the map. Instead, the factorization in equation (11) is used to separate the robot state from the map. The particle filter is only used for the robot's state, often x, y and orientation (θ) for a terrestrial indoor robot. The map for each particle is updated according to the occupancy grid mapping algorithm, with the position fixed at the position of the particle. This separation allows the occupancy grid algorithm to work with a guaranteed position, while still allowing for uncertainty in the robot's pose. By looking at the highest probability location we can determine the current best guess of the robot's position and the map. At each step, the set of N particles X_{t-1} is updated to X_t according to the following algorithm:

```
1:   for k = 1 to N
2:       $x_t^{[k]} \sim p(x_t \mid x_{t-1}^{[k]}, u_t, m)$
3:       $w_t^{[k]} = p(z_t \mid x_t^{[k]}, m)$
4:       $\forall n.\ l_{t,n}^{[k]} = l_{t-1,n}^{[k]} + \log \frac{p(m_n^{[k]} \mid x_t^{[k]}, z_t)}{1 - p(m_n^{[k]} \mid x_t^{[k]}, z_t)} - \log \frac{p(m_n)}{1 - p(m_n)}$
5:       $X_t' = X_t' \cup \langle x_t^{[k]}, m_t^{[k]}, w_t^{[k]} \rangle$
6:   endfor
7:   for k = 1 to N
8:       draw i from $X_t'$ with probability $\alpha\ w_t^{[i]}$
9:       $X_t = X_t \cup \langle x_t^{[i]}, m_t^{[i]} \rangle$
10:  endfor
```

Table 2. Occupancy Grid FastSLAM algorithm

Line 2 updates the set of particles by randomly choosing a new location for each one based on the previous location of the particle and the motion model. The sensor model is used to determine a weight for the new location based on the sensor readings in line 3. The primary difference from MCL occurs in line 4, where the occupancy probability of the map attached to the particle is updated according to the sensor readings used at the particle's new location. Of course, these maps are maintained via the log odds ratio as in section 2.1. Finally, in the loop of lines 7 - 10, the updated particles are resampled. N new particles are chosen randomly according to the weights, with replacement, to make the new particle set. Resampling has the effect of replacing the particle weight with the number of samples at a location. Thus, low probability locations die out while high probability locations gather enough particles that, on the next update, the correct location will be selected by the motion model in line 2.

5. Dynamic Maps in MCL

One drawback to localization with MCL is that it requires a static map of the environment. Sensor readings are compared with the expected values from the map and the comparison generates the probability of the robot's location. Errors in the map are partially compensated for by increasing the error that is assumed for the sensors. Another way to compensate for map errors is that the number of correct sensor readings will probably overrule incorrect ones. However, because MCL combines sensor error and map error, as map error increases, the allowable sensor error decreases until finally the algorithm fails and the map must be rescanned. Each error in the map is usually a minor matter for a localized robot; it is the combination of minor errors that can cause problems.

A localized robot rarely becomes mislocalized due to map errors, but this is not true of global localization, where the robot's initial location is unknown. Especially in symmetric environments, global localization can easily fail due to minor map errors that would be ignored by a localized robot.

The approach described in this section is based on the idea that if a robot is localized it may reasonably expect its sensor data to reflect the environment. If that is the case, then it should be possible to update the map according to the sensor data. If a known error in the map is fixed, then the robot will have a greater ability to deal with any subsequent errors. Since global localization may depend heavily on minor features, having an updated map can be a great benefit.

Violating the static map assumption and detecting changes allows localization to be more accurate and more robust to error. It also provides additional information that may be useful in planning the robot's activities. Detecting opening doors and moving objects makes path planning more reliable, because it will be based on a more accurate representation. Further, when a new opening into an unexplored are is detected, the robot can add the new region to the map. The dynamic map algorithm described here makes it far easier for a robot to be deployed long term in an environment where other agents, including humans, are present and making changes.

Dynamic maps for MCL can also be implemented by identifying binary objects, such as doors, and tracking their status using similar probabilistic methods [Avots et al. 2002]. There are several benefits of having explicit objects. Since an object consists of multiple cells that have the same probability, each scan provides more information about the object, allowing its state to be altered more quickly. Also, since most of the map is not dynamic, the probability of objects can be changed much more rapidly, since changes in the objects probably will not be able to change the map to make an invalid location match the sensors. However, explicit objects need to be manually defined before execution, adding to the work of defining maps. Since objects are binary, either present or absent, a moving object must be represented explicitly by creating a binary object at each possible location. With the dynamic maps described here, an object can appear anywhere without user interference. Finally, the method in [Avots et al. 2002] involves a different importance factor, which increases the runtime logarithmically in the number of objects, making it unsuitable for having each map cell dynamic.

Algorithms for simultaneous localization and mapping (SLAM) have the ability to localize the robot and generate the map simultaneously in real time [Montemerlo et al. 2002]. These algorithms are meant to dynamically alter the map in the same way as dynamic map MCL. Many of these methods use an algorithm which is guaranteed to converge to a correct solution. However, they suffer from the data association problem. On every sensor scan it must be possible to uniquely identify which feature of the map is responsible for each sensor reading. If this is impossible, then the guarantee of correctness does not hold. SLAM does not discover and use cell correlations, so the rate of update is slower if the map changes, since each cell must be considered independently. Further, SLAM involves significantly more processing than MCL, using up computing power that may not be necessary, especially after the map is generated. Dynamic map MCL was created specifically to provide an accurately changing map without incurring any significant overhead. Since it is a constant time addition to MCL, the map can be updated without requiring any more computing power than ordinary localization. Of course, the map cannot be generated from nothing as it can with SLAM, but once the map exists it can be kept up to date almost without cost. SLAM also, in common with ordinary MCL, makes the assumption that the map is static. Over time, the algorithm becomes more certain of the map and any changes will take longer to appear. Dynamic MCL explicitly makes the assumption that the map will change.

Algorithms that consider dynamic environments typically assume a static map with dynamic elements, such as people, which must be eliminated from consideration. In effect, these algorithms assume a static map but allow an additional form of sensor noise in the form of moving people. [Hahnel et al. 2003] describes a method for creating a map, using standard EM SLAM techniques, which can discover the static map of the environment despite dynamic elements. Similarly, [Fox et al. 1999] gives an algorithm for using MCL in an environment with many moving objects. Although both these papers give a method for handling a dynamic environment, they both assume an underlying static map. The benefit of dynamic MCL is that the static map assumption is no longer necessary. As the algorithm runs, it changes the map to correspond to the environment. Since dynamic MCL is implemented as an augmentation to ordinary MCL, there is no reason that other augmentations could not be used if warranted by the problem. For example, the algorithm described in [Fox et al. 1999] to discard readings relating to dynamic objects during MCL can coexist with my algorithm for modifying the map in accordance with changes in the environment. Dynamic MCL allows fundamental changes to be accounted for, as opposed to merely ephemeral objects that are only observed once.

5.1 Dynamic Maps

In order to alter the map, it needs to be added to the MCL formula. Consider each cell of the map to be an independent object, which can be either present or absent. Although independence is usually not entirely valid, it is an assumption that is often made. Consider $y_t = \{y_{1,t},\dots,y_{K,t}\}$ the set of individual cells in the map. Since we are considering these cells to be independent, if the location is known, then $p(y_t \mid x_t,z_t) = \prod p(y_{k,t} \mid x_t,z_t)$.
With this background, the new state equation is $p(y_t, x_t \mid z_t, u_t)$. Unfortunately, it turns out that this equation cannot be factored, since the map state is not fully determined with only the current location. However, notice that each sample in MCL represents not only a current location, but also the history of locations that lead to that location. Since each particle is only moved according to the motion model, they may be considered as x^t instead of x_t with no change to the algorithm. If we use the equation $p(y_t, x^t \mid z^t, u^t)$, then it is possible to factor it and we can also use the MCL algorithm without significant changes. The factorization used is similar to the one in [Avots et al. 2002], which was used to add the state of doors into the MCL algorithm.

5.2 Factoring

The size of the state space of (y_t,x^t) is exponential in the size of y_t, so we need some way of factoring the posterior in order to reduce the state space.
First, Bayes rule and the Markovian property give us:

$$p(y_t, x^t \mid z^t, u^t) = \eta p(z_t \mid y_t, x_t) p(y_t \mid x^t, z^{t-1}, u^t) p(x^t \mid z^{t-1}, u^t) \tag{12}$$

Now, consider the 3 parts of equation (12).
Without any data we assume that all states are equally likely, and also that the probability of a random sensor scan is a constant. Therefore:

$$p(z_t \mid x_t, y_t) = \frac{p(x_t, y_t \mid z_t) p(z_t)}{p(y_t, x_t)} = \eta' p(x_t \mid z_t) \prod_k p(y_{k,t} \mid x_t, z_t) \tag{13}$$

Remembering that cells in the map change status independently in the model, and again using the Markovian assumption, we get:

$$p(y_t \mid x^t, z^{t-1}, u^t) = \prod_k \sum_{y_{k,t-1}} p(y_{k,t} \mid y_{k,t-1}) p(y_{k,t-1} \mid x^{t-1}, z^{t-1}, u^{t-1}) \tag{14}$$

Finally:

$$p(x^t \mid z^{t-1}, u^t) = p(x_t \mid x_{t-1}, u_t) p(x^{t-1} \mid z^{t-1}, u^{t-1}) \tag{15}$$

Recombining these three equations and simplifying we get the factorization:

$$p(y_t, x^t \mid z^t, u^t) = p(x^t \mid z^t, u^t) \prod_k p(y_{k,t} \mid x^t, z^t, u^t) \tag{16}$$

Which contains the original MCL posterior and a new probability for the cells in the map. See [Avots et al. 2002] for more details about the factorization.

5.3 Binary Object Bayes Filtering

Since the method for calculating $p(x^t | z^t, u^t)$ is already known in the MCL algorithm, the only new method needed is to calculate the probability of each cell in the map. These cells are binary objects since they are either present or absent. Each $y_{k,t}$ can be either 0 or 1 with the probability of each summing to 1. Thus, the method for calculating the probabilities is the same as in [Avots et al. 2002]. Let $\pi_{k,t} = p(y_{k,t} = 1 | x^t, z^t, u^t)$. Then

$$\pi_{k,t} = \frac{p(y_{k,t}=1|x_t,z_t)\,p(z_t|x^t)}{p(y_{k,t}=1)\,p(z_t|x^t,z^{t-1},u^t)}\pi^+_{k,t} \tag{17}$$

where

$$\pi^+_{k,t} = p(y_{k,t}=1\,|\,y_{k,t-1}=1)\pi_{k,t-1} + p(y_{k,t}=1\,|\,y_{k,t-1}=0)(1-\pi_{k,t-1}) \tag{18}$$

In equation (17) the only unknown probability is $p(z_t | x^t, z^{t-1}, u^t)$ in the denominator. Rather than trying to calculate it, we exploit the fact that $y_{k,t}$ is binary so $(1 - \pi_{k,t})$ can be calculated in the same way as $\pi_{k,t}$ using $y_{k,t} = 0$ instead of $y_{k,t} = 1$. The two equations are then divided to cancel the unknown quantities.

$$\frac{\pi_{k,t}}{(1-\pi_{k,t})} = \frac{p(y_{k,t}=1|x_t,z_t)}{1-p(y_{k,t}=1|x_t,z_t)}\,\frac{1-p(y_{k,t}=1)}{p(y_{k,t}=1)}\,\frac{\pi^+_{k,t}}{\pi^-_{k,t}} \tag{19}$$

The result, equation (19), consists entirely of known quantities. $p(y_{k,t}=1)$ is the prior probability that a cell is occupied. The various $p(y_{k,t} | y_{k,t-1})$ values are the transition probabilities for a cell, $\pi_{k,t-1}$ are, of course, the prior occupancy probabilities and finally, $p(y_{k,t=1} | x_t, z_t)$ is the probability of occupancy given robot location and sensor data. To get a useful value from the odds ratio, we use the equality $\pi_{k,t} = 1 - (1 + \pi_{k,t}/(1 - \pi_{k,t}))^{-1}$.

The representation of $\pi_{k,t}$ is actually in closed form, so it requires only a constant time operation to calculate. Since $p(y_{k,t}=1 | x_t, z_t)$ involves sensor values and raytraces which are already used for MCL, little additional processing should be required. It is possible to modify the importance factor, as in [Avots et al. 2002], to take into account the new map data, where each cell is not merely present or absent but has a probability of presence. Using this data results in a runtime increase at least logarithmic in the number of binary objects. The probability of a location becomes the sum of the probabilities of that location for both states of all visible objects, multiplied by the probability of the object states. While that is acceptable if there are only a small number of objects, such as doors, if the objects are the cells of a map, the number becomes unmanageable. However, most map data used for MCL is actually represented as probabilities in an occupancy grid map, but is thresholded to be either present or absent. I decided to use the same simplification for my algorithm and consider each cell as either present or absent depending on a threshold value on its probability. The processing time therefore remains unchanged, since the importance factor is calculated in the same way.

5.4 Cell Correlations

In order to perform the factorization, it is necessary to assume that map cells change independently of each other. However, this assumption is not entirely accurate. In fact,

groups of adjacent cells that represent the same objects are likely to be completely dependent. To some extent ordinary MCL also assumes cells are independent, but it only becomes relevant when the cell probabilities are changed in dynamic MCL. It is easy to model correlations by annotating the map with correlation probabilities between adjacent cells, however, using this information is more difficult. Methods such as loopy belief propagation or variational methods [Jordan et al. 1999] can propagate belief through a connected graph, but they are time consuming and sometimes do not converge. Since dynamic MCL must run in real time without being much slower than ordinary MCL, these techniques are not sufficient. However, it should be noticed that the cell correlations in a map are of restricted types. Small groups of adjacent cells are highly correlated, while being uncorrelated with their neighbours. Because of the limited correlation, it is possible to use a modified variational technique in order to implement cell correlations. When a cell is updated, the update is propagated to adjacent cells along the links, but the propagation is not permitted to flow back to a cell that has already been modified. Also, the flow stops when the accumulated correlation probability falls below a threshold. In practice, only a few steps occur, but these achieve a significant improvement in the results.
The key to using cell correlations is to perform operations using two different and conflicting sets of assumptions. Each set of assumptions reduces one part of the problem to a solvable operation but makes the other part intractable. We have already seen that, by assuming cells to be independent, we can factor the belief as:

$$p(y_t, x^t \mid z^t, u^t) = p(x^t \mid z^t, u^t) \prod_k p(y_{k,t} \mid x^t, z^t, u^t) \tag{20}$$

This factorization is used to update the individual cells according to the robot's sensors. However, once the update is performed we discard both the assumption and the resulting factorization. Instead, we assume that each cell depends on its neighbours and is independent of the robot's sensors and position. According to this set of assumptions:

$$\begin{aligned} p(y_t, x^t \mid z^t, u^t) &= p(x^t \mid z^t, u^t)\, p(y_t \mid x^t, z^t, u^t) \\ &= p(x^t \mid z^t, u^t)\, p(y_t) \\ &= p(x^t \mid z^t, u^t) \prod_k p(y_{k,t} \mid y_{k-up,t}, y_{k-down,t} y_{k-left,t} y_{k-right,t}) \end{aligned} \tag{21}$$

The determination of the robot's position is unchanged, but the map cells now depend on their neighbours and not on the robot. By making this assumption any changes made to the map can be propagated to the adjacent cells and the weight of the cell correlations adjusted. Separating the algorithm into two phases with different assumptions allows the algorithm to consider additional dependencies without having to deal with the intractable problems caused by the interaction of the new dependencies with the old. In effect, during the first phase of the algorithm, as represented by equation (20), we assume that cells are influenced only by the robot, with additional effects coming from some unknown source. During the second phase, shown by equation (21), we assume that cells are only affected by their neighbours, with other changes caused by external, unconsidered, forces. Of course, two sets of contradictory assumptions cannot possibly be a reflection of reality, however, each

assumption is a reasonable simplification and using both sets iteratively results in less simplification than either set exclusively.
In dynamic MCL, it is necessary to modify the cell correlation probabilities dynamically on each cycle. However, given the nature of the sensors used, it is unlikely that adjacent map cells will be observed on a single scan. The solution to the problem is to cache observed changes to each cell until an adjacent cell has also been observed. At that point, the difference in the changes of the cells can be used to adjust the correlation between them.
Adding cell correlations significantly improves the dynamic MCL algorithm since a correlated group of cells can change together whenever any member of the group is observed. The result is that although the update of individual cells must be slow to allow localization to work, if a group of cells change they will update very quickly, since each observation will correlate them, and as they become more correlated every observation of a member of the group will update the entire group. Thus, an object can appear or vanish more quickly than any single cell.

5.5 Algorithm

The preceding formulae can be used to augment an implementation of MCL in order to modify the map dynamically during processing. The MCL algorithm must raytrace along all sensor paths to calculate the probability of a particle. However, if the robot's position is known with high probability, then any differences between the sensor reading and the raytrace are more likely to be errors in the map than in the sensors. In that case, the logical action is to correct the map.
The method I used is to consider each cell of the map to be present with probability $\pi_{k,t}$. On each step of the MCL algorithm an augmented raytracer is used for the robot's most likely location. The augmented raytracer follows a ray normally, passing through each map cell along the ray. However, at each cell along the path, the probability of that cell is altered according to equation (19). Although the augmented raytracer could be run on all samples, it is more productive to determine the most likely location and use the augmented raytracer only on it. When the robot's location is not known, the new raytracer is not used.
For calculating the sensor probability of each cell, the simplifying assumption that either that cell or the existing wall is correct is used. The assumption is necessary because the normalizer for the sensor probabilities is not known, so some method must be used to normalize the values. In practice, when a new cell becomes occupied, it exceeds the threshold before any other cell, and then the assumption becomes valid again. The short period during which it is invalid for some cells does not affect the operation of the algorithm.
In order to find the robot's most likely location, the sample with the highest importance factor is used. Other locations are possible, including the weighted average of all samples. The algorithm cannot run if the robot's location is unknown.
These implementation details do not change the fundamental algorithm, which is a implementation of MCL together with the binary object formulae as described above. The only simplification to equation (19) is in the calculation of $p(y_{k,t} = 1 \mid x_t, z_t)$, a value which is at best a numerical approximation to the error in a physical sensor device.
The following pseudocode summarizes the algorithm for dynamic MCL.

```
1:  Repeat N times
2:      Draw a random particle
3:      Move particle according to the motion model
4:      Annotate particle with a weight from the sensor model
5:  Resample a new set of particles from the annotated set
6:  Find the most probable location (mean of particles)
7:  For each sensor reading
8:      Raytrace to the nearest occupied cell
9:      For each cell on the path
10:         Alter the occupancy probability of the cell
11:         Alter the occupancy probability of neighbouring cells according
    to influence
12:         Mark cell as observed
13:         If neighbouring cell marked observed
14:             Adjust influence between cells
15:             Unmark cells as observed
```

Table 3. Dynamic MCL algorithm

5.6 Experimental Evaluation

The dynamic map algorithm was implemented and tested using real data collected in our building. The data was created using a Pioneer 2Dxe robot equipped with a laser rangefinder. The objective of the tests was to show that the map could be updated correctly without introducing errors or causing localization to fail. Since the algorithm has an almost constant runtime there is no tradeoff necessary between the time required to update the map and the benefit obtained by doing so.

Dynamic map MCL is designed to gradually update the map of the environment used for localization. Ordinarily, MCL uses a static map which, in a dynamic environment, gradually becomes less accurate. The experiments were selected to validate the dynamic algorithm by demonstrating that over time the map becomes a more accurate representation of the environment. Obviously, localization and global localization will perform better on a more accurate map. However, the improvement is a greater tolerance for other sources of error and is not detectable from the results of localization. The experiments demonstrate that the map is updated correctly, the benefit obtained from this update depends on the specific problem.

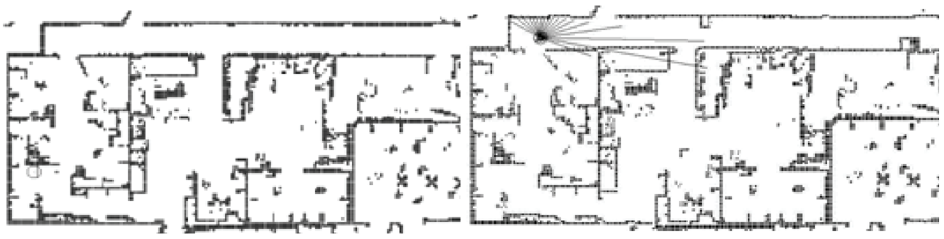

Figure 1. Before and after 2 passes through the environment

Figure 1 shows the map of the environment used to generate the test data. Changes were made to the environment after the map was scanned by opening and closing doors and by

placing boxes in the corridors. After 1 pass through the changed environment the robot has mostly added the new features to the map and has correlated the changed objects, allowing them to be completed very quickly.
After 2 passes, all changes have been completely added to the map. The rate of update is slower than in [Avots et al. 2002] because each cell must be observed several times, instead of each object. However, without correlations it takes at least 5 passes to completely adapt the map. Allowing cells to become correlated permits much faster updating without compromising localization. In [Avots et al. 2002] the dynamic objects can be updated in a single pass because they are manually defined ahead of time and are known to be completely correlated. Since dynamic MCL has no predefined objects or correlations, it is necessarily slower, but because it can discover the correlations it can still update very quickly.

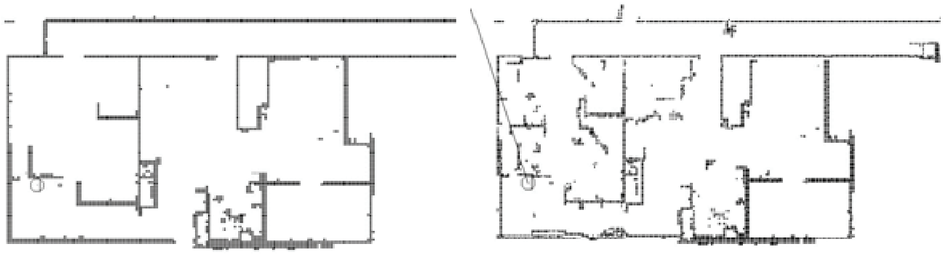
Figure 2. Before and after 5 passes through the environment using a schematic map

Another test, shown in Figure 2, was to use the same data but starting with a map consisting of the minimum possible information. From a schematic map consisting of only the walls and partitions, the algorithm was able to adapt it with all the features that were missing. Those portions of the map that were observed were corrected properly. The benefit of being able to start with a limited map is that it may not be necessary to scan a map manually with a robot. Instead, the map could be entered using blueprints of the environment and, as the robot passed through, it could correct the map until it was accurate. Usually, MCL uses the most accurate map possible, since it will lose accuracy over time, but with a dynamic map the accuracy of the map increases as the robot traverses the environment. Of course, portions of the environment that were insufficiently observed were not completely added to the map, so the result is not identical to the environment. However, observed areas have become more accurate and the map will only become a better reflection of the environment as the robot traverses it over time.
Another feature noticeable in Figure 2 is that some of the objects in the corridor are somewhat more diffuse than they appeared in Figure 1. Since the map is less accurate to begin with, localization is necessarily less accurate. As the map is corrected and localization becomes better, the location of the objects becomes clearer. After 5 passes, the objects are almost completely defined in the map, but some of them obviously require several more passes to full correct them. The benefit of dynamic MCL is that the robot can operate independently of this process. As it performs its task, the map becomes more accurate. All other data files tested exhibited similar behaviour, with the observed portions of objects being added to the map and no new errors introduced.

5.7 FastSLAM Comparison

The dynamic MCL algorithm is very similar to FastSLAM, with the major difference being that FastSLAM keeps the map state separately for each particle, while dynamic MCL maintains a single global map. The single map results in two significant changes in the behaviour of the algorithm. First, the run time over ordinary MCL is only increased by a constant in terms of the number of particles. Regardless of how many particles are necessary for localization, dynamic map MCL requires the same amount of additional processing. FastSLAM requires additional processing that is at least linear in the number of particles, disregarding the work necessary to continuously copy the maps. Dynamic MCL thus requires significantly less processing power than FastSLAM.

The per particle map is what allows FastSLAM to determine a map from nothing while localizing, since it can maintain multiple hypotheses until the robot observes a distinguishing feature. However, these map hypotheses necessarily include some unlikely maps. When the environment is mostly known these borderline maps are unnecessary. The basis of dynamic MCL is that the map is mostly known. In this case, the robot's position can be determined and the map can be altered based on the single correct position, instead of updating based on multiple hypothesized positions. In the case with a pre-existing map, FastSLAM's ability to update the map is provided by dynamic MCL without the drawback of having to consider maps based on multiple conflicting paths. Of course, if the map is unknown considering multiple paths is necessary for success, so dynamic MCL is in no way a replacement for FastSLAM, it merely uses similar ideas to apply to a situation that FastSLAM does not handle well. If the map of the environment is mostly known in advance, dynamic MCL provides an efficient solution to handling dynamic elements and previously unobserved areas, without causing additional uncertainty.

To discover if dynamic MCL provides appreciable efficiency gains over FastSLAM when the appropriate map is available, the FastSLAM algorithm was run on the same data set as in Figure 1. FastSLAM was able to generate a map, but it took 428 seconds and, of course, did not include the areas that were not visited. In contrast, dynamic MCL completed the 2 passes in 68 seconds, an 84% improvement. When only minor features need to be updated in a mostly complete map, it is unnecessary to incur the cost of FastSLAM, since in these cases dynamic MCL is far more efficient while providing the same result. Dynamic MCL also allows previously visited areas to remain in the map, even if the robot has not observed them.

6. Skeletal FastSLAM

While FastSLAM is a good solution for localization and mapping, it suffers from some problems, notably the loop closure problem. As the robot travels around a loop in the environment, it has no way to incrementally correct its position. Only once the robot arrives at the end of the loop can it realize that the correct path is the one that arrives in the right place so that the map joins up. Because particle filtering only represents the highest probability locations, over a long loop the correct path may be lost. Surviving this problem requires a number of particles relative to the size of loops in the map, which means the algorithm increases in runtime and memory with the size of the map.

SLAM is normally defined as generating a map from total uncertainty about the environment. However, there is often some information available about the map, especially in an indoor environment. Unless your robot is a bulldozer, it is constrained to follow certain paths indoors, corresponding to the building's corridors. These corridors are relatively easy to describe based on a simple floor plan, or even by observing the environment. In this section, I demonstrate how minimal information about the skeleton of the environment can be used to improve FastSLAM and reduce the loop closure problem, requiring only enough particles for the local areas of uncertainty. Skeletal FastSLAM provides an intermediate step between pure localization with a static map and pure SLAM with total uncertainty about the environment. Many problems with some preexisting knowledge might benefit from this approach.

The primary contribution of skeletal FastSLAM is to bridge the gap between the total knowledge required by MCL and the complete uncertainty that is the initial condition for SLAM. Although these two algorithms are powerful, there are many situations that are somewhere between the two conditions. For these problems, the choices are to accept the error caused by the uncertainty in MCL or to discard the initial information in SLAM. The algorithm described in this chapter allows FastSLAM to take advantage of some initial information. Similarly to the dynamic map MCL algorithm in Section 5, skeletal FastSLAM applies to problems with partial knowledge of the environment. The ability to use partial knowledge increases the usefulness of FastSLAM to situations that would ordinarily be much more difficult.

The key to using a skeleton map of the environment in FastSLAM is to realize that, especially in an indoor environment, the robot must follow certain paths. Obviously, a particle whose path corresponds to one of these corridors in the environment is more likely than one traveling at a tangent to the corridor. Of course, this only applies if the particle is close enough to the corridor, but when one of the corridors affects the robot's path, it can act as a very useful indication of the correct path.

6.1 Monte Carlo Localization with Paths

Although MCL is originally defined to solve for only the robot's current position, as in section 3 and [Dellaert et al. 1999], it is trivial, by recording the past poses of each particle, to alter it to track the robot's entire path. The derivation is similarly easy to alter, again using the Markovian assumption, producing models identical to ordinary MCL.

$$p(x^t \mid z^t, u^t) = \eta p(z_t \mid x^t) \int_{x_{t-1}} p(x^t \mid u^t, x^{t-1}) p(x^{t-1} \mid z^{t-1}, u^{t-1}) dx_{t-1} \tag{22}$$

$$\begin{gathered} p(x^t \mid u^t, x^{t-1}) = p(x_t \mid x^{t-1}, u^t) p(x^{t-1} \mid x^{t-1}, u^t) \\ p(x_t \mid x^{t-1}, u^t) = p(x_t \mid x_{t-1}, u^t) \\ p(x_t \mid x^{t-1}, u^t) = p(x_t \mid x_{t-1}, u_t) \\ p(x^{t-1} \mid x^{t-1}, u^t) = 1 \end{gathered}$$

$$p(x^t \mid z^t, u^t) = \eta p(z_t \mid x_t) \int_{x_{t-1}} p(x_t \mid u_t, x_{t-1}) p(x^{t-1} \mid z^{t-1}, u^{t-1}) dx_{t-1} \tag{23}$$

6.2 Derivation of FastSLAM with Skeleton

In order to consider a topological map in FastSLAM, we need to add it to the equations in a form that can be easily calculated. Let S be the skeletal map, then the FastSLAM factorization becomes:

$$p(x^t, m \mid z^t, u^t, S) = p(x^t \mid z^t, u^t, S) \prod_n p(m_n \mid x^t, z^t) \tag{24}$$

We assume the map is independent of the skeleton, which only affects the robot's position. Thus, the occupancy grid mapping portion of FastSLAM is unchanged. Only the localization needs to take S into account.

$$p(x^t \mid z^t, u^t, S) = \eta p(z_t \mid x_t) \int_{x_{t-1}} p(x^t \mid u^t, x^{t-1}, S) p(x^{t-1} \mid z^{t-1}, u^{t-1}, S) dx_{t-1} \tag{25}$$

Note that, because the map is independent of the skeleton, the skeleton does not affect the sensor readings z_t, which depend only on the robot's state, including the map.
The new motion model for localization is $p(x_t \mid u_t, x_{t-1}, S)$. However, it is not obvious how to sample from this model as required by MCL. Fortunately, given that the distance between x_t and x_{t-1} is small, we can factor the model into our original model and an additional term representing the motion probability of the motion given the skeleton map.

$$p(x^t \mid u^t, x^{t-1}, S) = p(S \mid x^t, u^t, x^{t-1}) p(x^t \mid u^t, x^{t-1}) / p(S \mid u_t, x^{t-1}) \tag{26}$$

Equation (26) can be greatly simplified using the Markovian assumption again. Also, notice that the same operation as in equation (23) produces the ordinary motion model in the second term, while still leaving the entire path for use with the skeleton map.

$$p(x^t \mid u^t, x^{t-1}, S) = \eta p(S \mid x^t) p(x_t \mid u_t, x_{t-1}) \tag{27}$$

Having factored the motion model from the skeleton, all that remains is to convert $p(S \mid x^t)$ into a form that can be calculated.

$$p(x^t \mid u^t, x^{t-1}, S) = \eta p(x^t \mid S) p(S) p(x_t \mid u_t, x_{t-1}) / P(x^t) \tag{28}$$

$$p(x^t \mid u^t, x^{t-1}, S) = \gamma p(x^t \mid S) p(x_t \mid u_t, x_{t-1}) \tag{29}$$

Putting equation (29) back into the localization formula of (25) results in localization which takes into account the skeleton map.

$$p(x^t \mid z^t, u^t, S) = \eta p(z_t \mid x_t) p(x^t \mid S) \int_{x_{t-1}} p(x_t \mid u_t, x_{t-1}) p(x^{t-1} \mid z^{t-1}, u^{t-1}, S) dx_{t-1} \tag{30}$$

The final equation indicates that the modification to the motion model alters the weight of each particle. Thus, the localization step continues as normal while the probability of the particle's motion based on the skeleton map is multiplied with the sensor probability to determine the likelihood of the sample. The result will be to make particles which travel according to the skeleton more likely to be resampled then those which conflict.

6.3 Creating the skeleton map

Of course, the first step in implementing this technique is to define what exactly is meant by a skeleton map. The skeleton is defined by a series of line segments marked by their endpoints. Each line segment marks a corridor that constrains the robot's direction of travel. It is not necessary for every possible path to be represented, only those composing major loops in the environment. The skeleton gives the direction and length of each corridor, including the structure of their intersections. Such a map is very easy to construct, especially in an indoor environment where a schematic diagram is often available. Also, buildings are usually constructed with corridors at right angles, making it easy to determine the intersections of the skeletal map. In such an environment, any user could easily define the necessary skeleton with minimal understanding of the underlying algorithm.

6.4 Defining the skeleton model

In order to actually implement skeletal FastSLAM as defined in equation (30), we need to create a method of calculating the model $p(x_t \mid S)$. Fortunately, this model does not need to be sampled from as does the motion model, allowing more flexibility in its creation. Since the objective is to more highly weight paths the closer they are to the corridor, some type of probability based on the difference in angle between the path and the skeleton is the obvious choice. The model used is a Gaussian distribution centered at 0 degrees mixed with a uniform distribution according to a gain value. The smaller the difference between the angle of the robot's path and the angle of the line segment of the map, the more probable the particle. Of course, this only applies as the particle travels along the particular corridor. If the robot turns away from the corridor it is probably exploring some area not represented by the skeleton. In that case, the probability is a uniform distribution. Also, the current segment of the skeleton map that the robot is following is determined by which segment is closest. However, if the distance between the robot and the line is too great, then the probability is once again uniform, since a particle must be within a corridor to be affected by it. The result is a model that increases the probability of particles traveling along the skeleton and decreases the probability of those traveling at a tangent to it, while leaving those that are not following the skeleton unchanged. The model for particles within range of a skeleton line segment is illustrated in Figure 3. However, note that the actual values depend on the parameters selected for gain, variance, and threshold angle.

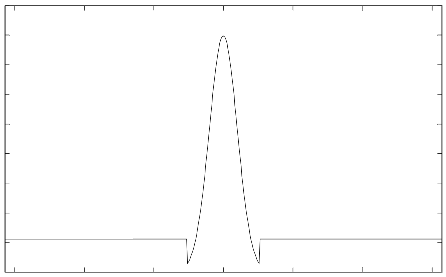

Figure 3. Skeleton map probability model

6.5 Algorithm

Given the derivation in section 6.2 and the model from 6.4, the actual implementation of skeletal FastSLAM is relatively straightforward. In order to reduce errors caused by minor corrections in the robot's heading while it follows a corridor, linear regression is used to track the line which best fits the robot's path. The regression is restarted every time the robot changes its current closest line segment. Then, the difference between the robot's course and the direction of the skeletal line segment is simply the difference between the angle of two lines, a straightforward algebraic computation. With that angle, the skeleton map model can be evaluated and the only change in the algorithm in Table 2 occurs on line 3, which becomes $w_t^{[k]} = p(z_t \mid x_t^{[k]}, m) * p(x_t^{[k]} \mid S)$.

It is, of course, necessary to provide various parameters, notably the variance and gain of the skeleton model as well as the threshold distance for the robot to be within a corridor and the threshold angle for the model. However, most of these parameters depend on the physical features of the environment and good values can be determined by examining its structure. The threshold distance depends on the corridor width, while the threshold angle depends on the relative corridor angles. Finally, the gain depends on how well the environment is represented by the skeleton map. These values probably do not need to change between different environments or robots, unless there are radical differences in the map. Even then, convergence will probably only require more particles, rather than failing.

Compared to ordinary FastSLAM, using a skeletal map adds runtime that is linear in the number of line segments in the skeleton. Since the topology of an indoor environment is usually fairly simple, the increase in processing required is minimal. If skeletal FastSLAM can reduce the number of particles required for convergence in an environment then the gains in processing time will more than offset the increase required to implement the algorithm.

6.6 Experimental Evaluation

In order to validate skeletal FastSLAM with occupancy grid maps it was tested against data sets gathered using a real robot in an indoor environment, as well as with various simulated data sets. The simulated data demonstrates the benefits of the algorithm in various situations, while the physical data sets show that it really does generate improvement in a real environment.

Loop closure is one of the major problems with FastSLAM and the skeletal algorithm is designed to reduce the processing required for loops in certain environments. The experiments were chosen to show the actual reduction provided by the skeleton in specific environments. From the results presented here we can determine that skeletal FastSLAM will provide a benefit in a wide range of circumstances where the fundamental assumption of fixed corridors applies. Since we cannot specifically test an algorithm’s loop closure ability, we rely on tests of the minimum processing required to develop a map with the correct structure as determined by a human observer. Although this criteria is somewhat vague, there was no problem in making the decisions since the maps tended to either converge correctly or diverge to random nonsense. The minimum number of particles necessary to generate a correct solution was used to determine the minimum run time for each algorithm. Since skeletal FastSLAM and ordinary FastSLAM require approximately the same amount of processing the skeletal algorithm must converge on fewer particles to provide a benefit. Comparing the minimum run times for convergence proves the benefits of using the skeleton do not outweigh the extra processing required to compare particles to the skeleton map.

6.6.1 Simulated data

Data sets generated from simulated environments test the basic behaviours of the skeletal algorithm in standard situations. One of the most basic environments is a wide, straight corridor. A 40 meter long corridor is typically a very difficult situation for FastSLAM because there is no indication as to the correct direction. A straight corridor gives almost exactly the same readings as one that curves slightly. Since the robot does not turn as it traverses the corridor there is no way for FastSLAM to correct the readings. Because of this, it required a minimum of 210 particles for ordinary FastSLAM to converge correctly using the data set. Compared to that, a single corridor is a very easy environment for skeletal FastSLAM, which required only 100 particles to converge. The run time for convergence of skeletal FastSLAM was 156 seconds for this data set, an improvement of 45% over regular FastSLAM's 283 seconds. Skeletal FastSLAM provides a serious advantage when an environment provides little information about global orientation.

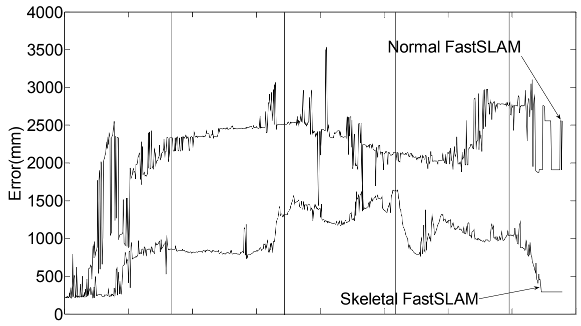

Figure 4. Error in position over time for normal FastSLAM vs. skeletal in a simple loop

The next data set was a large loop around a 40 meter square. Because the turns allow the robot to see back along its course, this environment was easier for FastSLAM to handle. Ordinary FastSLAM required 100 particles to converge with the data, while the skeletal algorithm was successful with 30. The time for convergence of 181 seconds for skeletal FastSLAM was a 67% improvement over ordinary FastSLAM's 558 seconds. In Figure **4** we can see the results of this test. For normal FastSLAM the error drifts, generally increasing over time, until the final section where the loop is closed. At that point, the error drops abruptly. In contrast, skeletal FastSLAM tends to retain a relatively constant error, changing only at the corners of the map, marked by the vertical lines, where the skeleton algorithm does not apply. The error is so much smaller when the loop is closed that it quickly decreases back to almost 0. Normal FastSLAM only managed to converge by shifting the entire map, thus retaining a larger error. By reducing the error increase in the corridors, skeletal FastSLAM is able to correct the position much more quickly when the loop is finally closed. The simulated data indicates that the algorithm provides a major benefit in the situations where it applies and leaves more leeway for handling the remaining situations, such as the corners.

6.6.2 Real data

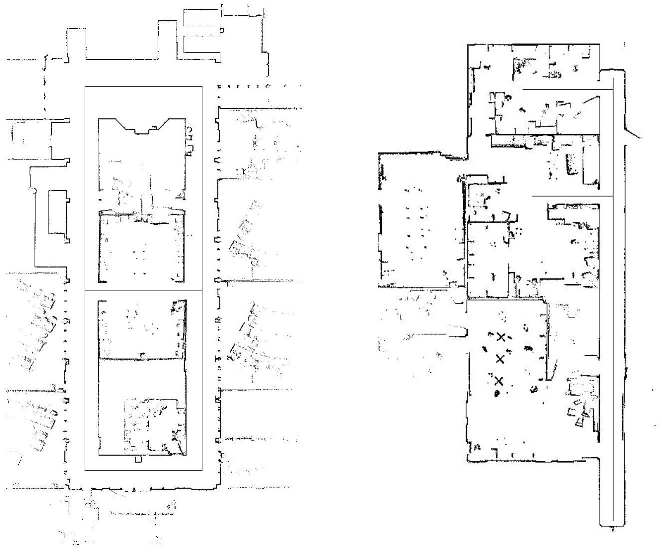

Figure 5. Two real environments with skeleton maps

Data from a 180 degree laser scanner mounted on a Pioneer 3Dxe differential drive holonomic robot was collected from two different real environments. Since there was no way to get the ground truth of the robot's position, I instead observed the minimum number of particles necessary for the map to converge to a representation which corresponded to the correct map. The primary observation about a correct map is that all of the loops are closed, connecting to the appropriate corridors. In practice, convergence was easy to determine, since, if the map did not converge, it instead diverged radically, becoming reduced to nonsense.

In the first environment in Figure 5, regular FastSLAM required at least 160 particles to converge, while using the skeleton map only required 100. The 60% larger set of particles for ordinary FastSLAM is necessary because the environment contains many long loops. Without the skeleton, more particles are necessary to allow these loops to close properly. The runtime of skeletal FastSLAM was a 58% improvement over the ordinary algorithm, converging in only 466 seconds compared to 737.

The second environment only has a single corridor and the robot travels through two rooms. Although the path into the rooms is marked by the skeleton, the path between them is not. The greater area that is not represented by the skeleton, coupled with fewer loops, results in less of an improvement. Skeletal FastSLAM needed 110 particles to converge in this environment, while ordinary FastSLAM needed 140, an increase of 27%. There was also an improvement of 23% in the runtime, with skeletal FastSLAM reducing the necessary time from 511 seconds to 391.

The improvements demonstrate that skeletal FastSLAM provides a significant improvement over ordinary FastSLAM, correctly converging with fewer particles, and thus less computation, using real data sets. Coupled with the simulated data, the results show that using a skeleton map is an effective addition to FastSLAM.

	Original		Skeletal		% improvement	
	particles	runtime	particles	runtime	particles	runtime
Simulated corridor	210	283.672s	100	156.329s	52.4%	44.9%
Simulated loop	100	558.078s	30	181.078s	60%	67.6%
Real environment 1	160	737.016s	100	466.938s	37.5%	36.6%
Real environment 2	140	511.047s	110	391.031s	21.4%	23.5%

Table 4. Experimental comparison of skeletal FastSLAM algorithm vs. original

6.7 Conclusion

Performing most tasks on a real robot, including path planning, requires knowing the configuration of the environment and the robot's position within it. One of the most adaptable representations is an occupancy grid map, which allows most types of environment to be accurately modeled. However, using occupancy grid maps limits the types of localization and mapping algorithms that can be used. Of the possible techniques, particle filter based methods are very effective. MCL provides accurate real-time localization while FastSLAM is a good solution for simultaneous localization and mapping. However, the basic implementations of these techniques have drawbacks in certain situations.

Section 5 describes an augmentation to MCL which allows the map to be updated according to the sensor measurements of a localized robot without a serious increase in running time. By considering each cell of the map to be an independent binary object and by making some simplifying assumptions, the static map required by MCL can be modified dynamically without requiring any user intervention. Instead of becoming less accurate over time, the map becomes more accurate as the robot traverses the environment. Experiments with real datasets show that the map can be updated properly without introducing errors. A change in the environment can be reflected in the map after very few passes by the robot. The result of the algorithm, having an accurate map, will always benefit the accuracy of MCL.

Dynamically correcting the map causes the largest source of error in MCL to decrease over time. Ordinarily, the best possible situation is for this error to remain constant, however in environments with dynamic elements, especially people, it is more likely that gradual changes occur. As the physical environment changes, errors build up in MCL, reducing its ability to handle any additional error. With dynamic updates the error is instead reduced over time, making localization more robust to other problems. Also, recognizing changes in the map might allow certain circumstances to be detected and considered in planning. For example, doors could be detected when they open and the robot could be sent to explore the new area. Also, new routes could be discovered as objects are moved. Removing the static map assumption greatly increases the power of MCL to handle real situations with dynamic elements. Furthermore, recognizing changes in the environment allows further improvements to be made at higher levels of control.

FastSLAM is an effective solution to both the online and full simultaneous localization and mapping problem in indoor environments where individual features are hard to determine. However, it suffers from problems in loop closure which require progressively more particles as the size of loops in the environment increase. By adding an easily created skeletal map into the algorithm, it is possible to significantly obviate this problem, allowing the FastSLAM algorithm to solve local uncertainties while aiding it in closing loops. A skeleton map indicates the direction that the robot must be taking so that, instead of wasting particles on multiple divergent trajectories, the algorithm can concentrate them around the correct path, significantly reducing the need for additional particles. As the corridors increase in length, ordinary FastSLAM requires an increasing number of particles, while skeletal FastSLAM continues to require only enough for the local uncertainties, becoming independent of the overall size of the map. Using a skeletal map is a low cost improvement to FastSLAM that is very useful in indoor environments whose overall configuration is known, even though the exact map may not be.

Skeletal FastSLAM, like dynamic map MCL, allows FastSLAM to handle situations with partial knowledge of the environment. Since initial knowledge no longer needs to be discarded, the behaviour of the algorithm is improved. By allowing additional information to be applied in the FastSLAM algorithm the technique can be very effective in specific situations where ordinary FastSLAM would require much more work. These methods of skeletal FastSLAM and dynamic map MCL lead to localization and mapping techniques that can generate a map and path from any type of starting information. Once the map and robot location are accurately known, it is possible to proceed with path planning and any other tasks the robot must perform.

7. References

Avots, D., E. Lim, R. Thibaux and S. Thrun (2002). A probabilistic technique for simultaneous localization and door state estimation with mobile robots in dynamic environments. *IEEE/RSJ International Conference on Intelligent Robots and System*, 2002. .

Dellaert, F., D. Fox, W. Burgard and S. Thrun (1999). Monte Carlo localization for mobile robots. *IEEE International Conference on Robotics and Automation*.

Eliazar, A. and R. Parr (2004). DP_SLAM 2.0. *ICRA*. New Orleans, USA.

Folkesson, J. and H. I. Christensen (2004). Graphical SLAM: A self-correcting map. *ICRA*.

Fox, D., W. Burgard and S. Thrun (1999). Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research* 11(3): 391-427.

Hahnel, D., R. Triebel, W. Burgard and S. Thrun (2003). Map building with mobile robots in dynamic environments. *IEEE International Conference on Robotics and Automation* (ICRA).

Jordan, M. I., Z. Ghahramani, T. S. Jaakkola and L. K. Saul (1999). An Introduction to Variational Methods for Graphical Models. *Machine Learning* 37(2): 183-233.

Julier, S. and J. Uhlmann (1997). A new extension of the Kalman filter to nonlinear systems. International Symposium on Aerospace/Defense Sensing, *Simulate and Controls*. Orlando, FL.

Lagarias, J. C., J. A. Reeds, M. H. Wright and P. E. Wright (1998). Convergence properties of the Nelder-Mead simplex algorithm in low dimensions. *SIAM Journal on Optimization* 9(1): 112-147.

Leonard, J. J. and H. F. Durrant-Whyte (1991). Mobile robot localization by tracking geometric beacons. *IEEE Transactions on Robotics and Automation* 7(3): 376-382.

Milstein, A. (2005). Dynamic Maps in Monte Carlo Localization. *The Eighteenth Canadian Conference on Artificial Intelligence*.

Milstein, A., J. N. Sanchez and E. T. Williamson (2002). Robust global localization using clustered particle filtering. *Proceedings of AAAI/IAAI*: 581–586.

Milstein, A. and T. Wang (2006). Localization with Dynamic Motion Models. *International Conference on Informatics in Control, Automation, and Robotics* (ICINCO).

Milstein, A. and T. Wang (2007). Dynamic motion models in Monte Carlo Localization. *Integrated Computer-Aided Engineering* 14(3): 243-262.

Montemerlo, M., S. Thrun, D. Koller and B. Wegbreit (2002). FastSLAM: A factored solution to the simultaneous localization and mapping problem. *Proceedings of the AAAI National Conference on Artificial Intelligence*: 593–598.

Moravec, H. P. (1988). Sensor Fusion in Certainty Grids for Mobile Robots. *AI Magazine* 9(2): 61-74.

Thrun, S., W. Burgard and D. Fox (2005). Probabilistic robotics, MIT Press.

Thrun, S., Y. Liu, D. Koller, A. Y. Ng, Z. Ghahramani and H. Durrant-Whyte (2004). Simultaneous Localization and Mapping with Sparse Extended Information Filters. *The International Journal of Robotics Research* 23(7-8): 693.