

Mapping Contoured Terrain: A Comparison of SLAM Algorithms for Radio-Controlled Helicopters

Cognitive Robotics, Spring 2005

Jason Miller
Henry de Plinval
Kaijen Hsiao

1 Introduction

In the context of a natural disaster, or when a military pilot has to eject in enemy territory, Search and Rescue teams often have to find people in unknown or hazardous areas. For safety reasons, Search and Rescue teams of the future will probably make use of unmanned aerial vehicles. For such a rescue vehicle, the ability to localize itself, both to avoid dangers (mountains/enemy bases) and to scan the entire area until survivors are found, is essential. However, this area may be unknown (enemy territory), or not mapped precisely (mountain summits). Moreover, global positioning systems (GPS) may not be usable in the area, or they may be not accurate enough, as is often the case in areas with dense foliage.

In such a situation, a helicopter capable of mapping its surroundings while localizing itself on this map would be of special interest. In this project, we have investigated such a platform. We have implemented a SLAM algorithm for a helicopter moving in an area with uneven terrain and using 3-D rangefinder sensors. Our goal was for this helicopter to be able to create a 2-D map of the ground surrounding it, while localizing itself on that map.

2 Goals of the Project

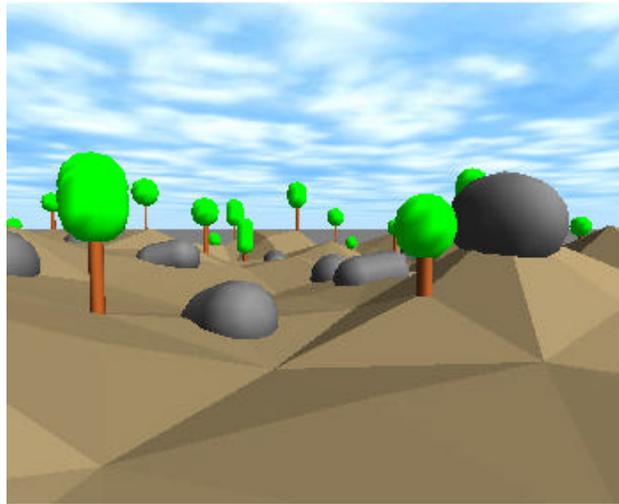


Figure 2.1: Problematic Terrain

The first goal of this project is to be able to do 2-D SLAM in a simulated, forested outdoor environment where the ground is not flat. Our platform of choice is a small, radio-controlled helicopter. In such a situation, traditional 2-D SLAM is problematic because the horizontal plane of the laser rangefinder can hit contoured ground, causing spurious landmarks to be placed on the map. The laser rangefinder can also miss low-lying landmarks if the helicopter is hovering too high. For instance, in the contoured scene in Figure 2.1, no horizontal plane of laser rangefinder casts can hit all the landmarks (rocks and tree trunks) at once, and the raised ground could be seen as a spurious landmark. Thus, we will use full 3-D laser rangefinder scans so that we can see all the landmarks in the scene. We will then process the 3-D scans to yield 2-D, leveled

range scans. Once we have 2-D, leveled range scans, we can use traditional 2-D SLAM algorithms to generate a 2-D map.

The second goal of this project is to compare two common SLAM algorithms and their abilities to perform 2-D SLAM in our environment. The two algorithms are FastSLAM, which uses landmarks as its map representation, and DP-SLAM, which uses occupancy maps. Both algorithms use particle filters to perform Bayesian updates. Each has advantages and disadvantages in terms of processing time, memory storage, and pre-processing requirements, and so our goal is to find out what the benefits and pitfalls of each method are, and to evaluate their performance and requirements.

The third goal of the project is to evaluate the ability of each 2-D SLAM algorithm to be extended to 3-D, by which we mean tracking the helicopter's pose as (x,y,z,θ) rather than simply (x,y,θ) . In general, we assume the helicopter is controlled to avoid significant pitch and roll, and so only yaw is considered. If we could track the helicopter's elevation using either the relative elevation of the landmarks or the elevation of the points on the occupancy map, we would have the full 3-D pose of the helicopter. With the full 3-D pose, we could create either 2-D terrain maps (by appending just the ground points to the determined path of the helicopter), or even full 3-D maps (by appending the full 3-D scans to the determined path of the helicopter).

Thus, the objectives of this project are:

- 1) To simulate an appropriate forested outdoor environment and the motion/perception of a small, radio-controlled helicopter
- 2) To segment and process the 3-D rangefinder data to create leveled range scans, rejecting spurious landmarks
- 3) To perform 2-D SLAM using the leveled range scan data with both landmarks and occupancy maps, and to compare these two algorithms
- 4) To evaluate the ability of 2-D landmark and occupancy map SLAM algorithms to be extended to 3-D

3 Previous Work

In terms of mapping of non-flat terrain from a helicopter, (Thrun, 2003) creates a 3-D map using 2-D rangefinder data. A small helicopter is equipped with rangefinders whose measurements lie in a plane perpendicular to the direction of motion. Using scan-alignment techniques, the noisy data is combined into a smooth 3-D picture of the world. However, they are not using SLAM, and the helicopter cannot image the same location twice with their algorithm.

(Montemerlo, 2003) creates a 3-D map of a non-flat underground mine from a cart platform. The robot uses a forward-pointing vertical rangefinder (whose plane is parallel to the direction of motion and the up-direction) to reject spurious 'wall' detections made by a horizontal rangefinder pointed at non-level ground. The resulting data is used to create a 2-D map with a normal 2-D SLAM algorithm. The 3-D map is then created using a plane of rangefinder measurements perpendicular to the direction of motion, combined using the helicopter's estimate of its location on the 2-D map generated using SLAM and smoothed using scan-alignment. This work is similar to what we are attempting to do, in that it performs 2-D SLAM with disambiguation of spurious

measurements due to contoured terrain. However, the leveled 2-D map they create uses only a single plane of vertical measurements. This is sufficient under the assumption that the world is reasonably rectilinear, consisting only of walls and ground. However, this is insufficient for outdoor environments.

Our project is essentially a combination of three papers. The first is (Brenneke, 2003), which uses a motorized cart equipped with a rotating 2-D laser rangefinder, just as described in our proposed project, to map a contoured outdoor environment. The paper describes how to use the 3-D cloud of points to disambiguate ground from landmarks in order to create a leveled 2-D range scan. The techniques we will be using to create our leveled range scans are largely similar to those used in this paper.

The second paper is (Montemerlo, 2002), on the FastSLAM algorithm that we will use as our landmark-based SLAM algorithm, and the third is (Eliazar, 2004), on the DP-SLAM 2.0 algorithm we will use as our occupancy map-based SLAM algorithm.

4 FastSLAM

The first SLAM algorithm we have implemented and analyzed is FastSLAM. FastSLAM uses the principle of particle filtering to explore several different hypotheses about the location of the robot and the map at the same time. It requires some pre-processing of the data, since the obstacles are all assumed to be point landmarks.

In FastSLAM, a set of particles is used to keep track of the position of the helicopter and the positions and uncertainties of the landmarks. Each particle contains a guess on the position of the helicopter, and the position and uncertainty for each landmark it has observed so far.

The algorithm is divided into four steps:

- *Motion Step*: as the helicopter moves, the particles update its position based on the motion inputs.
- *Data Association*: in most real-life applications, the computer ignores the number of the landmark it is observing (Is it one it has seen before? Which one? Is it a new one?). The data association algorithm determines, for each particle, which landmark it has observed.
- *Kalman Update*: for each set of observations and for each particle, the algorithm updates the position and uncertainties of the landmarks.
- *Resampling*: each particle is weighted based on its capability to predict the observation that was made. A particle that predicts the observation that was made with high probability gets a high weight. Then, the particles are resampled based on those weights: the best particles are copied, whereas the bad ones are deleted.

The following sections present these steps in detail.

4.1 Motion

Each time step, the helicopter moves in reaction to the inputs it is given. Those inputs are available to the FastSLAM algorithm. However, some uncertainty is added to the inputs reported to the algorithm, so that the helicopter motion does not fit perfectly

with the inputs. The different particles of FastSLAM sample the possible locations where the helicopter might have moved, given the inputs. This is done by applying the dynamic model to the position of the helicopter, and having each particle sample from the probability distribution function of the resulting helicopter position after motion. Figure 4.1 shows the particles sampling the possible locations of the helicopter after it has moved.

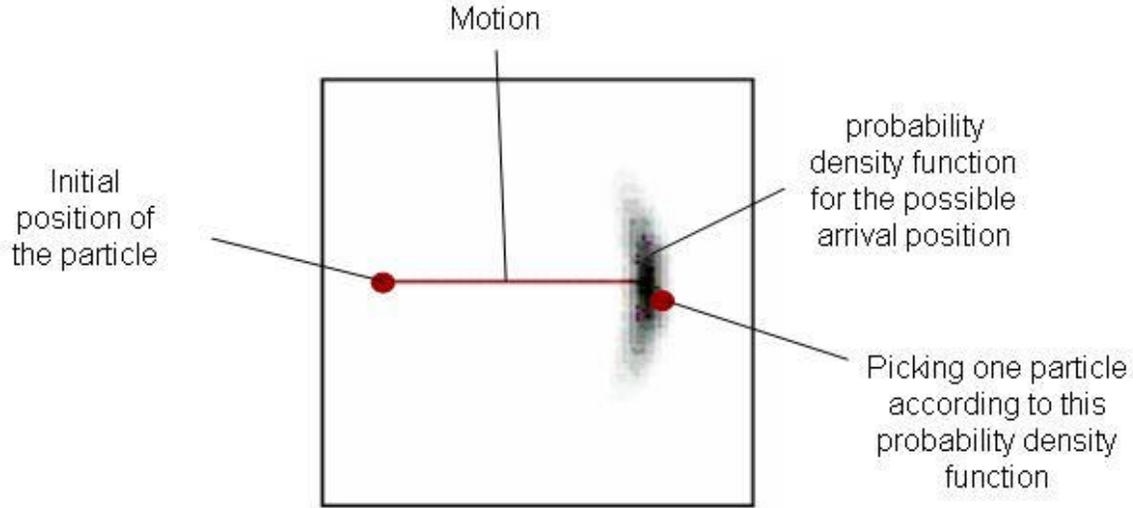


Figure 4.1: Propagation Step

In our program, the input consists of a rotation angle defined in the horizontal plane (again, the helicopter is assumed to be controlled in a manner that avoids pitch and roll, and thus we consider only yaw), and the 3D vector of its translation in its own frame of reference. To this ideal model is added Gaussian noise, both in rotation and in translation. The motion model is given by the following equations:

$$\begin{aligned}
 \mathbf{q}(t+1) &= \mathbf{q}(t) + \Delta \mathbf{q}(t+1) + \mathbf{e}_q(t+1) \\
 x(t+1) &= x(t) + \cos(\mathbf{q}(t+1)) * \Delta x(t+1) - \sin(\mathbf{q}(t+1)) * \Delta y(t+1) + \mathbf{e}_x(t+1) \\
 y(t+1) &= y(t) + \sin(\mathbf{q}(t+1)) * \Delta x(t+1) + \cos(\mathbf{q}(t+1)) * \Delta y(t+1) + \mathbf{e}_y(t+1) \\
 z(t+1) &= z(t) + \Delta z(t+1) + \mathbf{e}_z(t+1)
 \end{aligned}$$

Essentially, the motion consists of a rotation in the horizontal plane and a translation in the helicopter's frame of reference.

4.2 Data Association

In our implementation, the helicopter carries a 3-D laser rangefinder. The only information available to the helicopter is the distance at which the laser hit something in any given direction. However, since FastSLAM is based on keeping track of landmark locations, an algorithm must decide which landmark the laser has hit at each time step. This problem is called the data association problem: is each observation associated with a

landmark we have seen before or a new one, and if it is one we have seen before, which one?

For each particle and for each landmark this particle has already seen, the algorithm computes the probability that the current observation is of this landmark. This is done by computing the distance from the landmark to the observed obstacle, and using the uncertainty on the landmark position to calculate how likely it is that this distance could have been obtained by observing the landmark. The algorithm also computes the probability that the observed obstacle is a new landmark, which is essentially the probability that the observation did not belong to the most likely landmark.

Figure 4.2 shows a sample data association problem. The observed obstacle is very far from L2, so it is unlikely to be that obstacle. On the other hand, the observed obstacle is close to L1, which has a pretty high uncertainty. Thus, it is likely that the observed obstacle is L1.

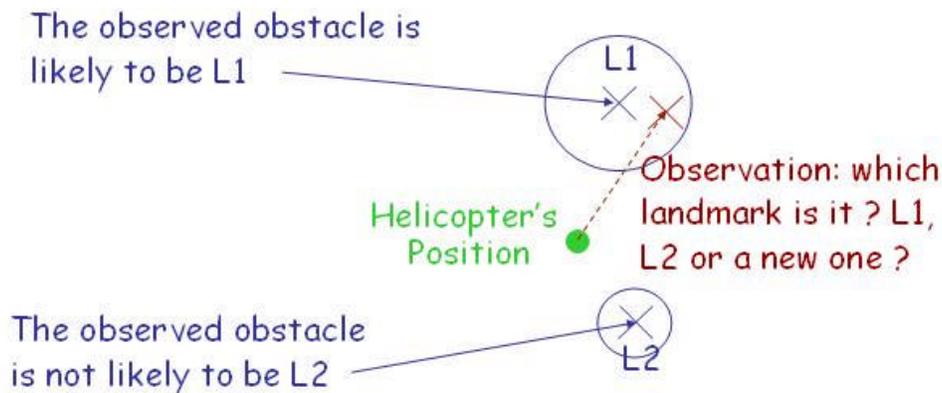


Figure 4.2: Data Association

For each particle, the weight for a landmark (which is proportional to the probability that this landmark is the one we are observing) is given by:

$$w = e^{-d^2/\mu^2}$$

where d is the distance between the landmark and the observed obstacle and μ is the uncertainty (standard deviation of the Gaussian) associated with that landmark.

$$\mu = \sqrt{P + \left(\frac{\mathbf{m}_x}{\sqrt{N}}\right)^2 + \mathbf{s}^2 + \mathbf{m}_x^2}$$

where P is the covariance of the landmark, \mathbf{m}_x is the uncertainty of the laser rangefinder observation, N is number of laser casts that were averaged to obtain the observed location of the landmark, \mathbf{s} is the standard deviation of the laser casts that hit the landmark, and \mathbf{m}_x is the uncertainty on the robot position.

The probability that the observed obstacle is a new landmark is then given by:

$P = \text{erf}\left(\frac{\mathbf{d}_{\min}}{\sqrt{2\mathbf{m}_{\min}}}\right)$, where erf is the integral of the Gaussian, \mathbf{d}_{\min} is the minimum distance to the landmarks, and \mathbf{m}_{\min} is the uncertainty of the corresponding landmark.

4.3 Kalman Update

The observations performed by the helicopter are used to update the positions and uncertainties of the landmarks. To accomplish this update, a Kalman Filter is applied to the landmark chosen by the data association algorithm. Since the landmarks are supposed to be motionless, the Kalman Filter is reduced to one single step: the Update Step.

Figure 4.3 shows a situation where the Kalman Filter is applied. From the observation and its uncertainty, the position and uncertainty of the landmark are modified.

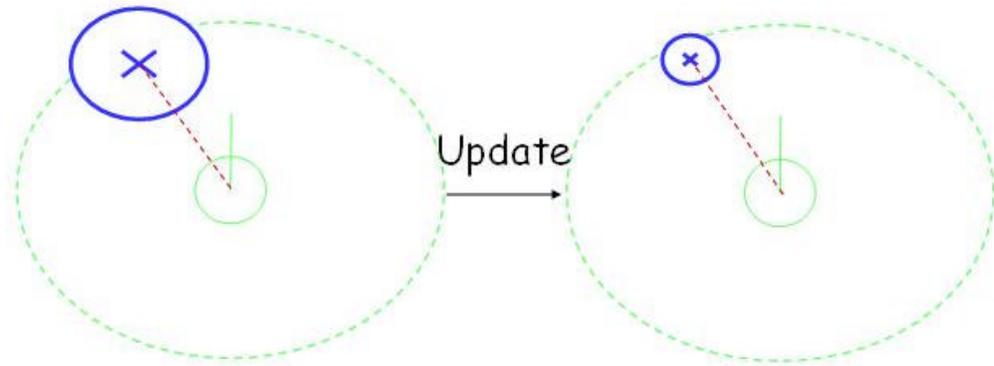


Figure 4.3: Kalman Update

The equations for the Kalman Update, since the landmarks are assumed to be motionless, are:

State Update: $\hat{\mathbf{x}}(t+1) = \hat{\mathbf{x}}(t) + K(t+1) * (y(t+1) - h(\hat{\mathbf{x}}(t)))$

$$K(t+1) = P(t)H(t+1)^T (H(t+1)P(t)H(t+1)^T + R(t+1))^{-1}$$

Covariance Update: $P(t+1) = (I - K(t+1)H(t+1)^T)P(t)$

In these equations, $\hat{\mathbf{x}}(t)$ is the state (position of the landmark considered). $K(t+1)$ is the Kalman gain, defining the confidence the algorithm has in the estimate and in the measurement. $y(t+1)$ is the measurement at time (t+1). H is the measurement function, and $H(t)$ is its Jacobian at time t . $P(t)$ is the covariance (uncertainty) at time t .

4.4 Resampling

After each time step, the algorithm replicates the ‘good’ particles, and gets rid of the ‘bad’ particles. For each particle, the probability of making the observation that was made is computed. This probability is the product of the probabilities of observing each obstacle that was actually observed. To compute the probability of observing a given obstacle, the algorithm finds the closest landmark, and computes the probability of making the observation that was made given that it is the observed landmark.

With this scheme, however, the particles can produce a new landmark each time an observation is made, and will thus have a very high probability of making the observation. To avoid the creation of excess landmarks, we add a penalty: the more landmarks a particle has, the lower its weight.

Figure 4.4 represents the resampling algorithm. Before resampling, the particles are spread out, because of the motion step. After, only the good particles are kept by the algorithm.



Figure 4.4: Resampling the particles

The weight for each particle is basically the probability of making the observation that was made. Thus, it is the product over the observations of the probability of making each of the observations. This probability is, as before:

$$w = e^{-\frac{d^2}{m^2}}$$

where: d is the distance between the landmark and the observed obstacle and μ is the uncertainty.

Then, to avoid allowing a particle to create excess landmarks to fit the observations, we include a penalty for having too many landmarks: the weight of each particle is divided by the number of extra landmarks it has in its line of sight, above the number of observed obstacles. For instance, if a particle has 5 landmarks in its line of sight, and the helicopter has observed 3 landmarks at this time step, the weight of this particle is divided by $(5-3) = 2$.

5 DP-SLAM

5.1 Introduction

Many SLAM algorithms (including FastSLAM) use landmarks to represent the world that the robot is trying to map. This can be very efficient since the state of the world is condensed down to a relatively small number of key points. However, in real environments, it can be very difficult to successfully identify and disambiguate landmarks from sensor data. Furthermore, the idealized representation of landmarks as single points in space does not always correspond well to the reality of three-dimensional space where objects have a non-zero size and may appear different from different angles.

To avoid these problems, some SLAM algorithms (including DP-SLAM) use an occupancy map to represent the environment rather than landmarks. An occupancy map is simply a discretization of space into a regular array. Each point in the array can then be marked as either “empty” or “occupied.” Figure 5.1 shows an example 2-D world and an occupancy map representation of it. By using an occupancy map, it is no longer necessary to find specific landmarks. When an observation (such as a laser range-finder measurement) of the world is made, the map can be consulted to determine whether an object was expected in the observed location. In essence, an occupancy map creates a regular structure of simple landmarks that are very easy to observe.

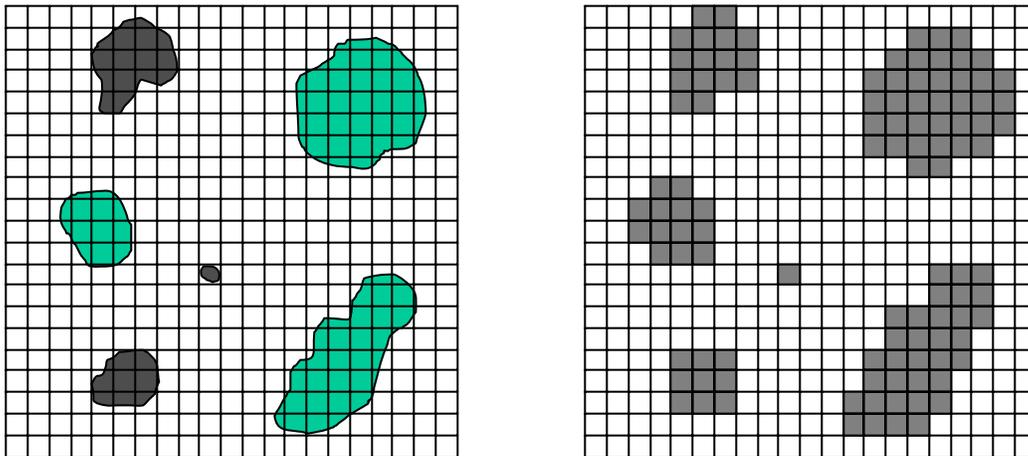


Figure 5.1: Overhead view of world with objects and its corresponding occupancy map representation.

However, using occupancy maps with particle-filter based SLAM provides its own set of new problems. One approach is to make each particle represent only the state of the robot (*e.g.*, position and angle) and have all particles share a single occupancy map. The difficulty with this is that different particles may need to make conflicting or inconsistent updates to the map. A more robust approach is to have each particle store its own map in addition to the robot state. Then, each particle will remain internally consistent and particles whose maps do not accurately match the real world will simply

be culled during resampling. This allows the algorithm to maintain multiple different conflicting hypotheses about the world until it is able to resolve the ambiguity by additional observations. However, from a practical standpoint, storing and manipulating a complete map for each particle can be extremely expensive, both in terms of storage and computation.

An additional problem with using occupancy maps is the fact that the discretization of the world creates an imperfect representation of it. In the example above, we have discretized both the shape and transparency of each object. Each grid square is considered to be either completely full (opaque to whatever sensor we are using) or completely empty. In reality, the area represented by each square will most likely be only partially filled or filled with something that is not consistently observed by the sensors. Figure 5.2 shows examples of how these two situations can cause problems in a situation where a laser range-finder is used for observations. If the laser enters the square from as show in (a) and hits the rock, this square appears opaque. However, if the laser were to pass through the same square from the angle shown in (b) it would appear transparent. Finally, if the rock were replaced with something like a tree (c) which has many small leaves that can move in the wind, the laser may penetrate to different depths on different observations, even if the observation angle is the same.

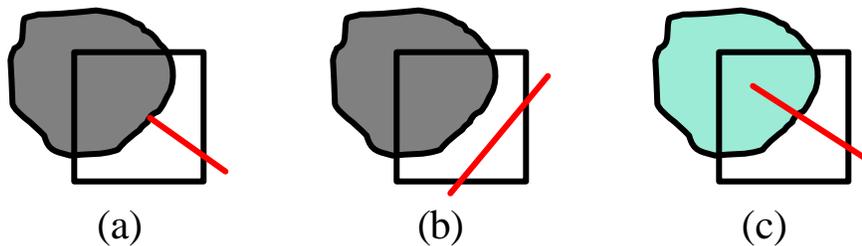


Figure 5.2: Differing observations due to inaccuracy of discretization

The DP-SLAM algorithm is an attempt to eliminate these problems and make occupancy map SLAM practical and robust in challenging environments. The first published version of DP-SLAM [2] (referred to by the authors as DP-SLAM 1.0) addresses the problem of map storage and manipulation. DP-SLAM 2.0 [3] builds on DP-SLAM 1.0 by adding a probabilistic occupancy model. Both of these innovations will be described in more detail below.

5.2 Algorithm

The basic structure of the DP-SLAM algorithm is the same as a standard particle filter SLAM algorithm. First, particles are propagated according to the motion model. Then, observations are made and used to update each particle's map and calculate its weight. Finally, particles are resampled probabilistically according to their weights. The process is then repeated for the next time step. The differences lie in the way that DP-SLAM represents the state of the world. Rather than using a Kalman Filter on landmark positions, DP-SLAM uses probabilistic occupancy maps. Since the core algorithm is

fairly standard, and was described in detail in the FastSLAM section, only the unique portions of the algorithm will be described in detail here.

Although it is desirable for each particle to have its own occupancy map, the burden of storing and copying these maps can be enormous. In particular, during the resampling phase of the particle filter algorithm, particles are selected based on their weights and then copied to form the next generation of particles. Since each particle contains its own map, it too must be copied to the new particle. Because several new particles can be created from a single old particle, the data must actually be copied rather than just reassigned to the new particle. This is particularly wasteful considering that many particles are storing the same data about parts of the map that have not been observed recently or never observed at all.

To address this, DP-SLAM uses only a single map where each square in the map is actually a tree containing observations for different particles. When observations are made, each particle inserts its updated data into the appropriate trees, tagged with the particle's unique ID number. Thus, no work is wasted copying or modifying squares that are outside the current range of observation. As long as a balanced tree structure (such as a Red-Black Tree) is used to store the observations, the time required to insert a new observation will be $O(\log P)$, where P is the number of particles being maintained.

However, the price for this efficiency in memory utilization is that retrieving data from the grid is significantly more complicated than a simple array access. When a particle needs to retrieve the data for a specific square, it first searches the appropriate observation tree for data tagged with its own ID number. If none is found, it doesn't necessarily mean that this particle doesn't know what's in that square. It may just mean that this particle has not changed that square and therefore inherited the value from the particle it was resampled from. This earlier particle is called an *ancestor* of the current particle. Therefore, the particle must next search the tree for data tagged with its ancestor's ID. If none is found, it must search using its ancestor's ancestor's ID and so on until it finds a value or runs out of ancestors (indicating that it knows nothing about that square). Thus a particle must keep track of its lineage to enable it to retrieve the most recently updated value for a given map square.

There are two problems with this. First, the lists of ancestor particles will continue to grow with each time step, thereby imposing a limit (due to memory exhaustion) on the number of time steps that can be executed. Second, storing these ancestry lists is inefficient because they must be copied during resampling and again, different particles resampled from the same ancestor will have largely the same list. Since each particle may be resampled by multiple "child" particles, it makes sense to maintain the ancestry in a tree as well. Each node in this tree is a particle, with the current batch of particles being the leaves. If each particle maintains just a pointer to its parent, we have no duplication of data when multiple particles are sampled from the same parent. Since each particle's map is actually stored in the observation trees, very little extra memory is required to keep older particles (which normally would have been deleted) in the tree. To ensure that the ancestries form a single tree, all the particles in the first generation are treated as though they were sampled from a single root particle. However, the size of this tree can still grow unbounded as we add new generations of particles.

To keep the ancestry tree manageable, two maintenance operations are required. First, particles that are not selected for copying during the resampling phase (and therefore have no children) may simply be deleted. If this causes the particle's parent to become childless, it may also be deleted, and so on, up the tree. When a particle is deleted, its observations are also deleted from the observation trees. To accomplish this efficiently, each particle must store a list of all the map squares that it has updated. Second, if a particle in the tree has only one child, the parent and child may be merged into a single node that takes ownership of the observations from both particles. This is accomplished by removing all of the observations from one of the nodes and reinserting them using the ID number of the other node. The lists of updated cells are then merged and the node can be deleted. Running these two maintenance steps after each resampling ensures that the ancestry tree has exactly P leaves and a minimum branching factor of two. This means that the depth of the tree is $O(\log P)$ and the total number of nodes in the tree is less than $2 \cdot P$. Thus the size and depth of the tree are dependent only on the number of particles, not the amount of time the algorithm has been running.

It is not immediately obvious from the above description of this algorithm that it is more efficient than simply copying complete maps. Although it is too lengthy to present here, the two DP-SLAM papers present a thorough analysis of the algorithmic complexity. They are able to show that DP-SLAM is asymptotically far superior to simple copying in the common case where the portion of the map observed at each time step is a small fraction of the total map.

Using the above tree-based algorithms, it is possible to efficiently maintain separate occupancy maps for each particle. The rest of this section will explain what data is stored in the maps and how it is used to calculate particle weights. Because laser range-finders are the typical choice for making observations for SLAM, the remaining discussion will focus on them.

To address the problems mentioned earlier related to partially filled or semi-transparent map squares, DP-SLAM 2.0 introduces a probabilistic occupancy model. Rather than marking each square as either "full" or "empty," DP-SLAM attempts to estimate the probability that a laser passing through the square will be stopped and the distance to the square will be returned by the range-finder. This is accomplished using a parameter \mathbf{r} for each square. The papers refer to \mathbf{r} as the "opacity" of a square but, based on the way it is used, it might be more appropriate to call it the "transparency." In other words, the higher the value of \mathbf{r} a square has, the less likely it is that a laser passing through that square will be stopped. The probability of stopping should also be related to the distance the laser travels through a square. If the laser barely nicks the corner of a square, it is less likely to be stopped than if it traveled all the way across the square. Therefore, the probability that a laser will be stopped while traveling a distance x through a square with opacity \mathbf{r} is defined as,

$$P_c(x, \mathbf{r}) = 1 - e^{-x/\mathbf{r}}$$

Using this value, the probability of any path that a laser takes through a series of squares can be calculated. Since the laser always travels in a straight line, the paths will,

in reality, always be rays originating at the helicopter position and with a length given by the range-finder.

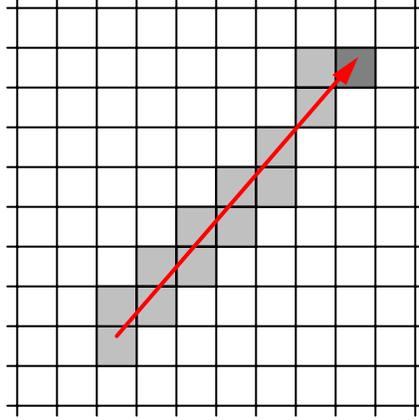


Figure 5.3: Example laser cast. Light gray squares are merely passed through. The dark gray square is where the laser stopped.

Intuitively, we calculate the probability that the laser will *not* be stopped in each square that the laser merely passes through, and the probability that it *will* be stopped in the square that terminates the path. The product of these probabilities is then the probability of the complete laser path. If the squares along the path are numbered in the order that the laser encounters them from 1 to i , then the probability that the laser will have traveled that path and stopped at square i is,

$$P(\text{stop} = i) = P_c(x_i, \mathbf{r}_i) \prod_{j=1}^{i-1} (1 - P_c(x_j, \mathbf{r}_j))$$

The last factor to account for is the laser error model. First, the distance \mathbf{d}_i from each square along the laser path to the end of the path is calculated. Using the laser error model, the probability that the range returned was off by \mathbf{d}_i , $P_L(\mathbf{d}_i | \text{stop} = i)$ can be calculated. As in the papers, laser error was assumed to be Gaussian with a mean of zero and standard deviation equal to the range-finder “accuracy.” Then we calculate the probability that the laser actually stopped in each square along the path and multiply it by $P_L(\mathbf{d}_i | \text{stop} = i)$. The sum of all these probabilities is the total probability of a given measurement,

$$\sum_{j=1}^i P_L(\mathbf{d}_j | \text{stop} = j) \cdot P(\text{stop} = j)$$

To calculate the complete weight for a particle, we take the product of the probabilities of all its measurements.

To estimate \mathbf{r} for each square, two parameters are maintained: d_t , the cumulative distance that lasers have traveled through the square and h , the number of times that the laser was observed to stop in the square. Thus for each laser measurement, the path of the laser through the grid is traced. At each square, the distance the laser travels through the square is calculated and added the value of d_t already stored there. For the final square in the path, the value of h is incremented. The estimated value of \mathbf{r} is then $\hat{\mathbf{r}} = d_t/h$.

5.3 Implementation

As with FastSLAM, we implemented DP-SLAM in Matlab. Although Matlab excels at dealing with vectors and matrices, it does not include native support for the tree data structures we required. Fortunately, we were able to find a free Matlab toolbox (Keren) that implements Red-Black trees. This toolbox makes use of a Matlab class called a pointer (Zolotykh). The pointer class is what allows nodes to refer to each other as parents or children. We were able to use the Red-Black tree structures directly for the observation trees in the occupancy map. However, since the ancestry tree is less structured, we used the pointer class directly to allow each particle to refer to its parent in the tree.

According to the DP-SLAM papers, each node in the ancestry tree (*i.e.*, the particles) needs to contain a unique ID number, a pointer to its parent node, a list of the map squares it has updated and, at least for the current particles, the robot pose. We found that this was insufficient information to perform the pruning and merging operations required to maintain the tree. In particular, pruning and merging both require some knowledge about the number of children a node has. We considered adding pointers to each node for its children but realized that this was excessive since the tree is only ever accessed from the leaves to the root and all that was required was a simple count of the children. Therefore, a field was added to each particle indicating how many children it has in the ancestry tree. This count is incremented as children are added during resampling and decremented as nodes are removed during pruning.

We also found that the algorithm given for merging nodes with their single children was not practical given the data stored in each node. It is only a minor point but the paper suggests moving observations from the child node to the parent and then deleting the child. This would require all the children of the child to have their “parent” pointers updated to point to the parent. Since nodes do not store pointers to their children, it is impractical to find and update them. We found that it was much simpler (and gave the same result) if the observations were transferred from the parent to the child and then the parent was deleted. In this case, only the “parent” pointer for the child needed to be modified to point to its parent’s parent. Since parent pointers are stored, it is trivial to find a node’s grandparent.

Although using the Data Structures and Pointer libraries saved us considerable time and effort, we were somewhat frustrated by the poor performance of the pointer class. Considerable energy was expended determining why tree accesses were so slow and attempting to work around the problem. Eventually, we determined that accessing a value using a pointer was two orders of magnitude slower than a native access. By

carefully reorganizing data and caching values to minimize pointer access, we were able to cut the run time for DP-SLAM in half. However, we were unable to eliminate all accesses and profiling suggests that a more efficient pointer implementation could cut the run time in half again. The real lesson learned is that Matlab is not a good choice for implementing tree-based algorithms. Both in terms of speed and memory usage, it is far inferior to C/C++ when dealing with tree data structures.

One important aspect of this algorithm that is not covered in detail by the papers is determining which grid squares a laser cast passes through and what distance it travels through each. Fortunately, this is a standard problem in computer graphics and can be efficiently solved using the Digital Differential Analyzer (DDA) algorithm. Our implementation of this algorithm is general enough to work with rays and grids of any dimensionality greater than or equal to two. (Of course the dimensionality of the rays and grids must agree.)

6 3-D Helicopter Simulation

The simulated world used to collect 3-D rangefinder data was created in Open Dynamics Engine, an open-source physics engine. However, no physics simulation components were actually used; only the collision detection system was needed for our application. The movement of the helicopter is calculated for each step based on its current location and the inputs from a human driving the helicopter on-screen, and the helicopter is teleported to the next location for the next laser rangefinder sweep. As you can see in the picture of the simulated world in Figure 6.1, the world consists of a bumpy terrain, rocks, trees, and the helicopter. The white spheres in the picture are the locations on terrain, rocks, and trees hit by the helicopter's laser rangefinder, which sweeps out a 3-D pincushion of laser casts at intervals of 5 degrees horizontally and 5 degrees vertically.

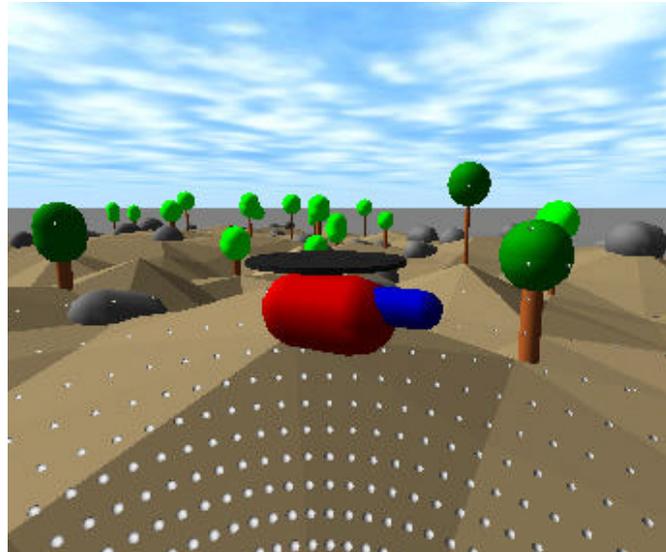


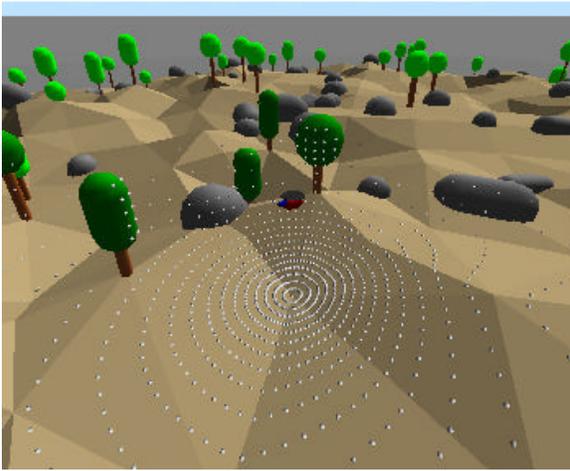
Figure 6.1: The Simulated World

The types of data recorded from the simulation are as follows: movement inputs to the helicopter, distances reported for each laser rangefinder cast, and for simplification of the task of segmenting landmarks from ground, the type of object hit by each laser rangefinder cast. In (Brenneke, 2003), the task of segmenting ground from objects of interest above ground is done in real-world situations. In that case, landmark points are defined as laser cast points that have at least one other point directly below them within the same vertical scan (and thus are vertical surfaces). This technique is good for picking out tree trunks, pillars, and fences, but it is not as good for picking out objects such as rounded rocks. In our simulation, we use both tree trunks and rocks for landmarks, and so a proper segmentation technique for our system would require slightly different heuristics. Also, our simulated rangefinder's resolution is not as good as the one used in (Brenneke, 2003), which has a resolution of 1 degree, due to the large computational requirements needed to maintain such high resolution in the simulation. Thus, to ensure that even landmarks with only one point per vertical scan may be used, we use additional information provided by the simulation (the type of object hit) to simplify segmentation of the laser data.

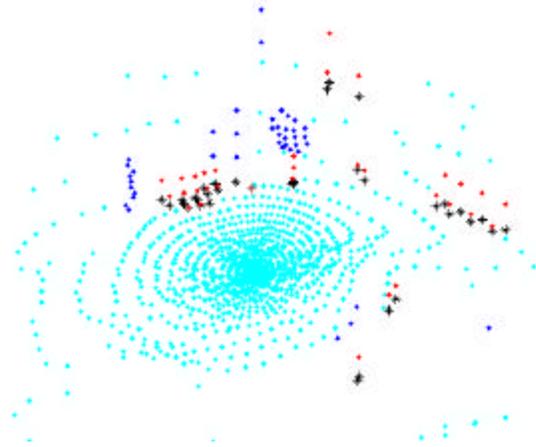
7 Leveled Range Scans

In order to perform 2-D SLAM on contoured ground, we needed to use 3-D laser rangefinder scans to avoid spurious landmarks from hilly ground and to see landmarks that would not appear in a 2-D scan. However, in order to use the 2-D SLAM algorithms, we need 2-D sensor inputs. This is where leveled range scans come in. A leveled range scan is a 3-D scan that has been squashed to a 2-D scan, with all the relevant landmarks and obstacles included.

In order to create a leveled range scan, we must segment the 3-D point cloud, as described above. Landmark points, which in our case include those hitting tree trunks and rocks, are separated from ground points and unwanted overhang points such as tree tops. The landmark points are then squashed to the level of the nearest ground points in their vertical scans, so that the z-position of the resulting landmarks is the height of the ground around the landmark. While the z-positions are not used for 2-D SLAM, they are used for 3-D SLAM, in which the elevation of the helicopter is tracked; we will discuss extensions to 3-D later on. As you can see in Figure 7.1, the scene on the left is turned into the segmented point cloud on the right. Cyan points represent ground, red points are landmark points, blue points are ignored treetop points, and black asterisks are squashed landmark points.



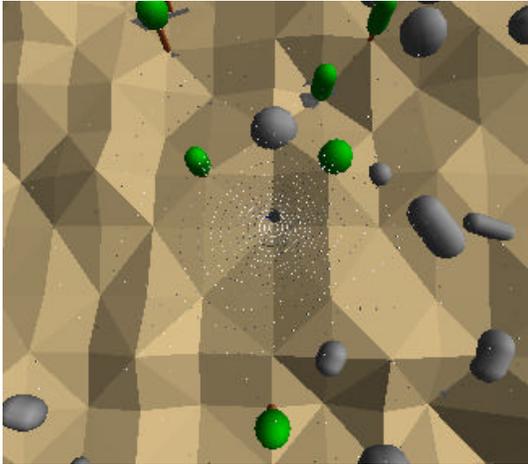
Scene From Helicopter Simulation



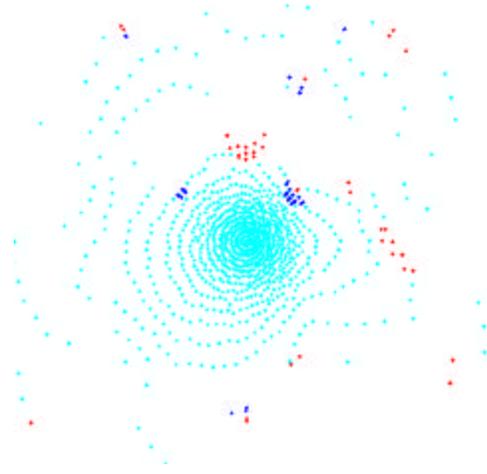
Segmented Point Cloud

Figure 7.1: Segmented Point Cloud

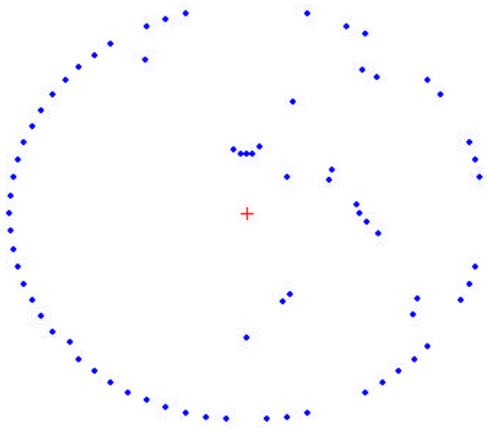
To obtain the leveled range scan from the segmented point cloud, we simply ignore all the z -values. The 2-D leveled range scan and the corresponding overhead view from the simulation are shown in Figure 7.2. The 2-D leveled range scan must then be processed into the relevant inputs for the particular 2-D SLAM algorithm. For occupancy map SLAM, the relevant inputs are the distances obtained by a 2-D laser rangefinder. Thus, we find the first landmark point hit by each horizontal set of laser scans, and return those values as if the world had been flat and a 2-D rangefinder were used. The resulting SLAM input is graphed in the lower left corner of Figure 7.2. For landmark SLAM, the relevant inputs are the locations of landmarks within range of the helicopter's rangefinder. However, landmarks can be large objects such as rocks. In order to obtain point locations for each landmark, we average the locations of the points hitting a single landmark. We then input to the SLAM algorithm the average location of each landmark, as well as the number of points that went into the average and their standard deviation (for uncertainty calculations). The resulting inputs to the 2-D landmark SLAM algorithm are shown in the bottom right corner of Figure 7.2, with circles denoting three standard deviations around the average landmark location.



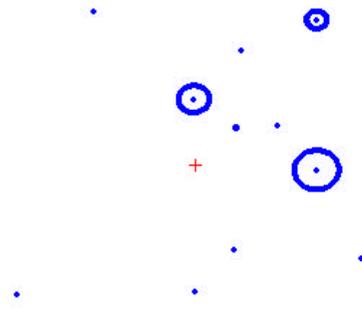
Overhead View From Helicopter Sim



Leveled Range Scan



2-D Laser Casts for Occupancy Maps



Landmarks with 3*std circles

Figure 7.2: Leveled Range Scan and Corresponding 2-D SLAM Inputs

8 FastSLAM Results

8.1 Leveled 2-D Final Maps

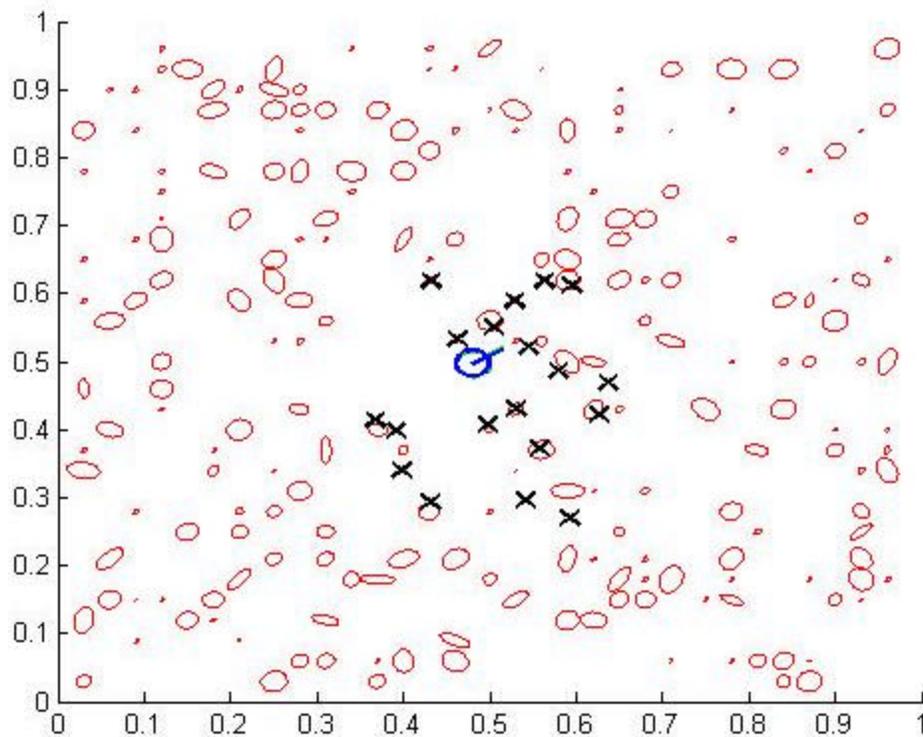


Figure 8.1: Final map for FastSLAM with 50 particles

Figure 8.1 displays the final map obtained by leveled 2-D FastSLAM with 50 particles, after 20 time steps. The red shapes represent the actual locations of the rocks and trees. The green circle is the actual position of the helicopter, and the line extending from the circle denotes the helicopter's orientation. The blue circle on top of the green circle is the estimated position of the helicopter. The black crosses are the estimated positions of the landmarks (trees and rocks).

As you can see, we have created a fairly large world for the helicopter to explore. However, because DP-SLAM takes an excessively long time to run in Matlab, we could only run for a limited number of time steps. Thus, the helicopter sees only the center of the map during the brief run. Figure 8.2 is the same map, but zoomed in. As we can see on this map, the result is reasonably accurate—the black crosses are on the rocks and trees they correspond to. Since the sensor is a laser rangefinder, the algorithm only has access to the front edge of each obstacle. Thus, the cross is seldom the center of an obstacle, since the helicopter never goes all the way around any of the rocks.

The large size of some of the obstacles (the rocks) was a large source of inaccuracy for FastSLAM. This is because as the helicopter flies, it sees each rock from a different perspective, and the laser casts that hit the rock average to a new location.

Thus, as the helicopter moves, the nearby rocks appear to move with it. This sometimes causes a great deal of confusion while localizing the helicopter. It also is a systematic source of error that is not accounted for by our model, which assumes that the landmarks are point locations that do not move. Furthermore, while we somewhat take into account the size of a rock by giving the algorithm an indication of the spread of laser cast points hitting the rock, there is no way to entirely account for the size and shape of a large rock. The rocks in our world are capped cylinders of varying lengths and at varying angles in the ground, and thus seeing a long rock end-on gives no indication of its length. This leads to inaccurate estimates of uncertainty in the position of the landmark, sometimes leading to spurious new landmarks.

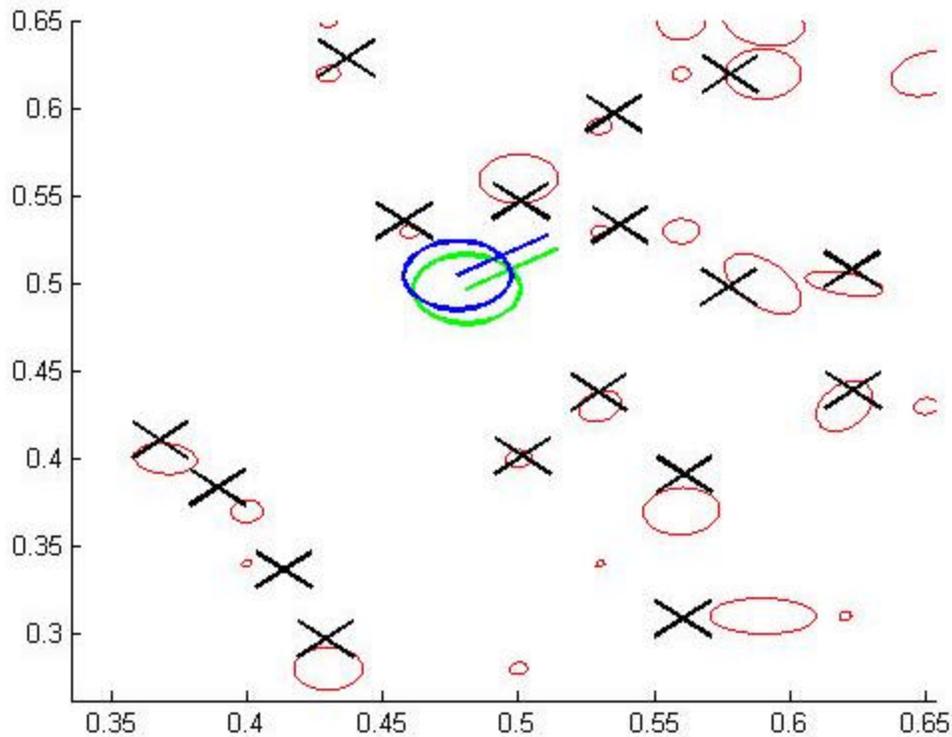


Figure 8.2: Final map for FastSLAM with 50 particles, zoomed in.

8.2 Paths

Figure 8.3 displays the actual and estimated helicopter paths. As one can see, they are fairly close to each other, showing that the algorithm is reasonably accurate. Figure 8.4 shows the same thing, but with the two sources of noise reduced: the motion and sensor noise is ten times less. As one can see, the error in position is greatly reduced.

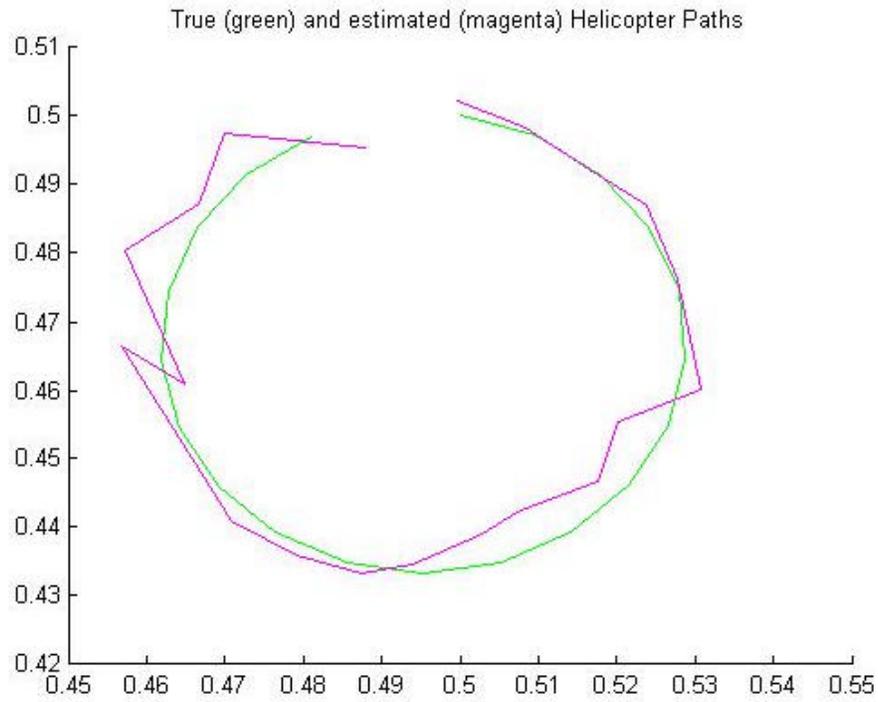


Figure 8.3: Actual and estimated helicopter paths.

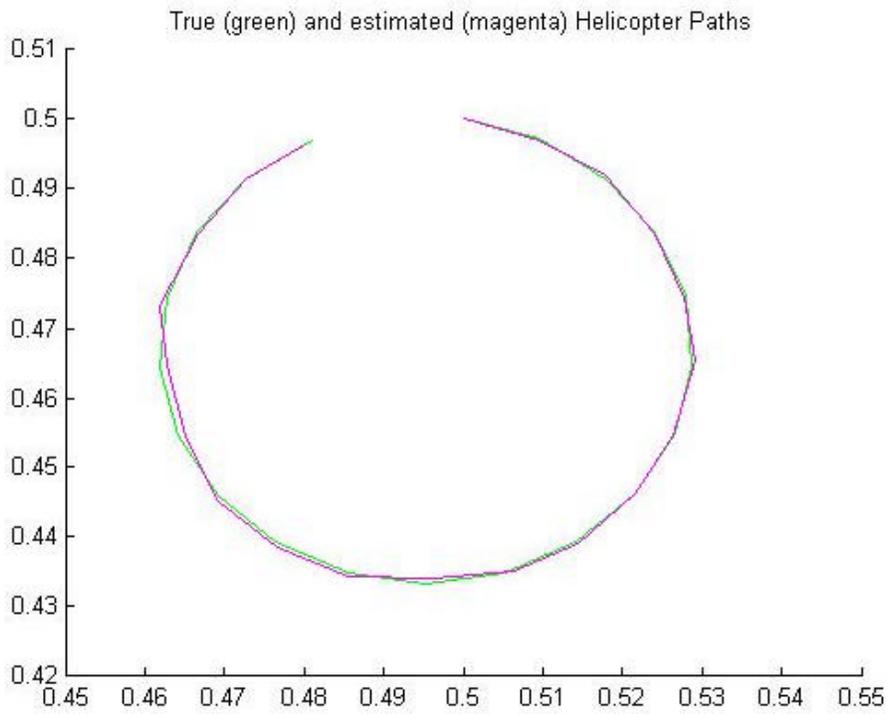


Figure 8.4: Actual and estimated helicopter paths (motion and laser noises divided by 10).

8.3 Speed

The amount of calculation time per time step required to run FastSLAM for varying numbers of particles is shown below in Figure 8.5. The time estimates include time for preprocessing the 3-D rangefinder data to yield landmark locations, which takes about 0.1 sec. As you can see, the time per time step increases linearly with the number of particles used. This is as one might expect. The accuracy of the results did not rise significantly with more than 50 particles, which takes only about 3 seconds per time step to run. This is already acceptable for most robotic SLAM algorithms. If the implementation had been done in C rather than in Matlab, the time would be greatly reduced.

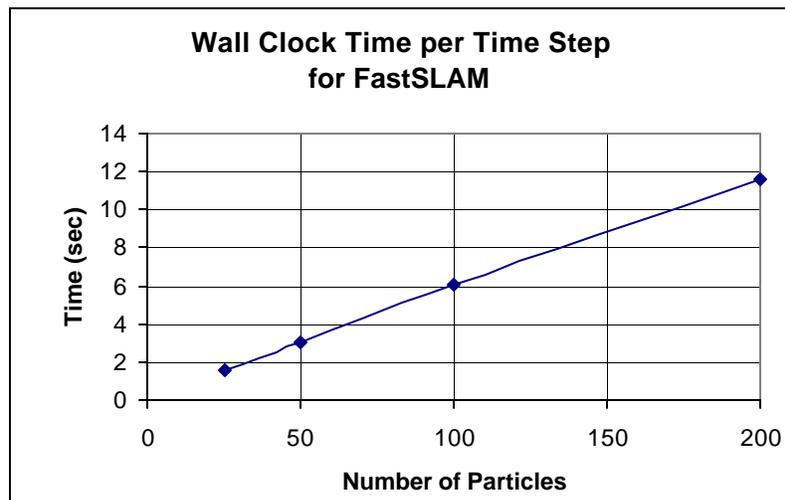


Figure 8.5: Time needed to run FastSLAM

8.4 Memory Usage

Figure 8.6 shows the total memory used while doing FastSLAM for varying numbers of particles. This amount includes an estimate of the amount of memory needed to do preprocessing of one time step's worth of data to find landmark locations. As you can see, the amount of memory needed also varies linearly with the number of particles used. For 50 particles, only 2 MB was required; again, if the implementation were done in C rather than Matlab, this number would probably be significantly less.

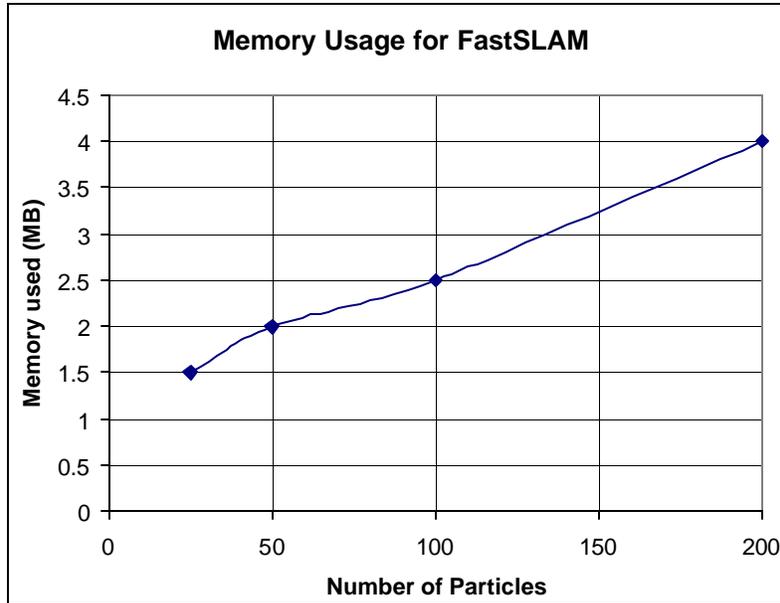


Figure 8.6: Memory Usage for FastSLAM

8.5 3-D FastSLAM

So far, the algorithms we have compared use a “leveled 2-D” approach: the observations' z -values are ignored, and a 2-D SLAM algorithm is used to generate a 2-D map. Now we will talk about doing FastSLAM in 3-D. In environments where the robot's elevation does not change significantly, one can create a 3-D map of the environment by appending 3-D scans to the path determined by a 2-D SLAM algorithm, as in (Montemerlo, 2003). The test run we did had the helicopter hovering at the same level throughout, so this would even have been possible with our data. However, if the robot's elevation can change significantly, which it often does with a helicopter platform, one must track the robot's elevation in order to create a 3-D map of the environment.

As mentioned in the section on creating leveled range scans, the landmark points are squashed to the level of the nearby ground, so that the position of a tree landmark is viewed as a point at the base of the trunk. This processing was done for use in 3-D FastSLAM. The 2-D FastSLAM algorithm extends easily to 3-D. In 3-D FastSLAM, the helicopter's pose is tracked as (x, y, z, θ) rather than simply (x, y, θ) , and the landmark positions are (x, y, z) instead of (x, y) . The Kalman Filter updates are changed to add the new dimension appropriately, and the rest of the algorithm is the same. The computational difficulty of doing 3-D with FastSLAM is only incrementally more than that of doing 2-D. This is one advantage of using FastSLAM over DP-SLAM, since, as we will discuss shortly, doing DP-SLAM in 3-D requires a prohibitive amount of additional computation.

Figure 8.7 represents a 3-D map obtained after 20 time steps. Again, the red circles are the actual positions of the obstacles. The green and blue circles are the actual and estimated positions of the helicopter, respectively. The black crosses represent the

estimated positions of the observed landmarks. The size of the crosses represents the uncertainty on their positions.

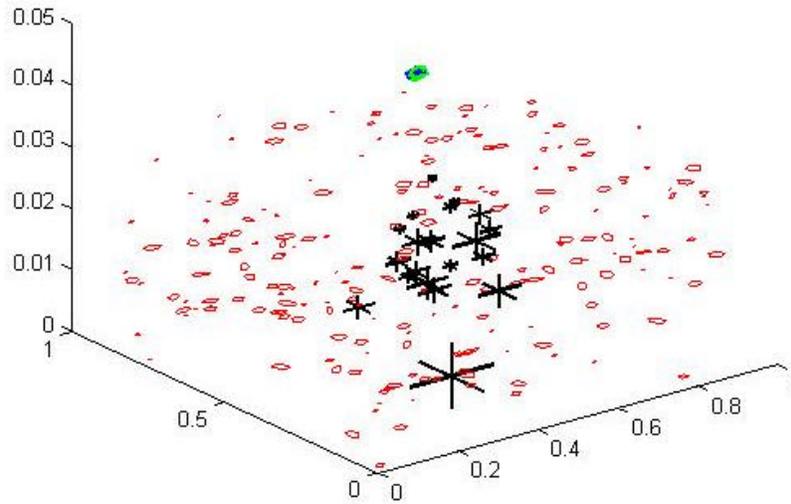


Figure 8.7: 3D map of the world obtained by 3DFastSLAM after 20 time steps.

To make the final map clearer, Figure 8.8 represents the final map obtained by 3-D FastSLAM, to be compared with the map in Figure 8.2. As we can see, the accuracy is very comparable. The time per time step is also very similar to that obtained for 2-D SLAM (3.5 sec for 50 particles).

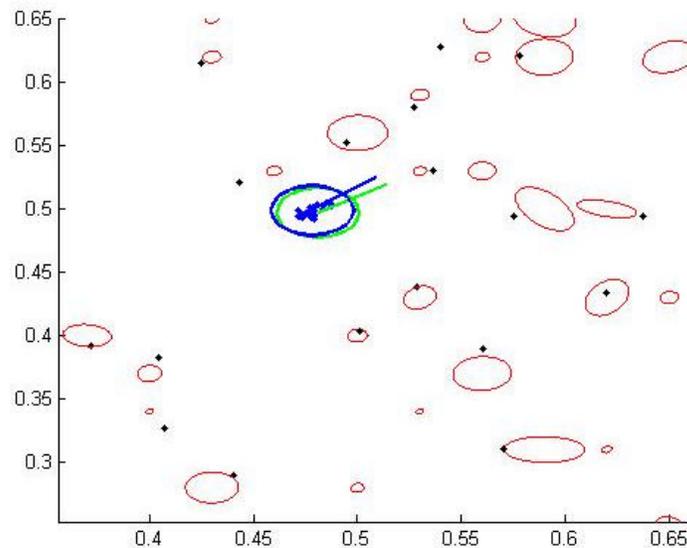


Figure 8.8: Final map for 3D FastSLAM (the map is seen from top)

Figure 8.9 represents the actual and estimated squashed paths from 3DFastSLAM, with the same noise parameters as the 2-D equivalent in Figure 3. As you can see, the accuracy is comparable.

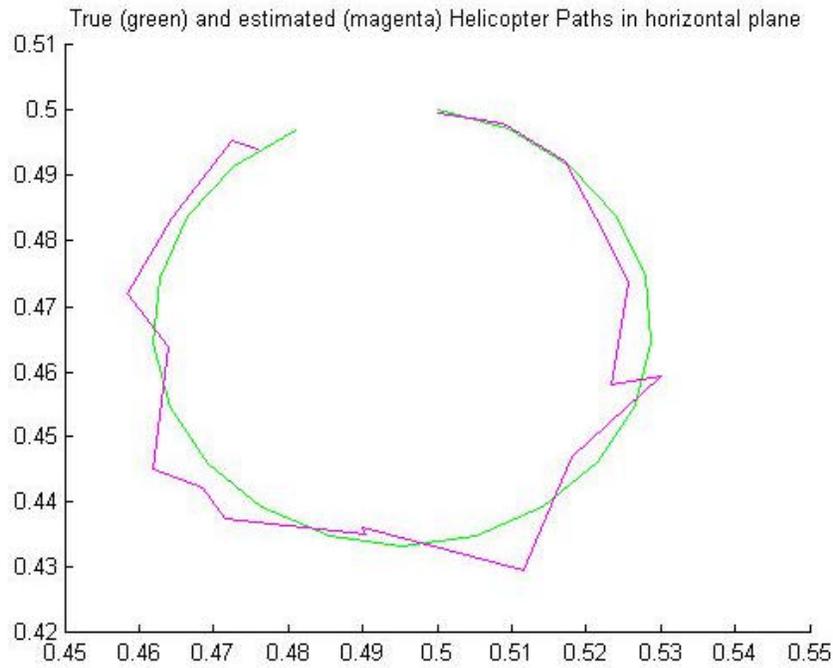


Figure 8.9: Squashed actual and estimated paths from 3DFastSLAM.

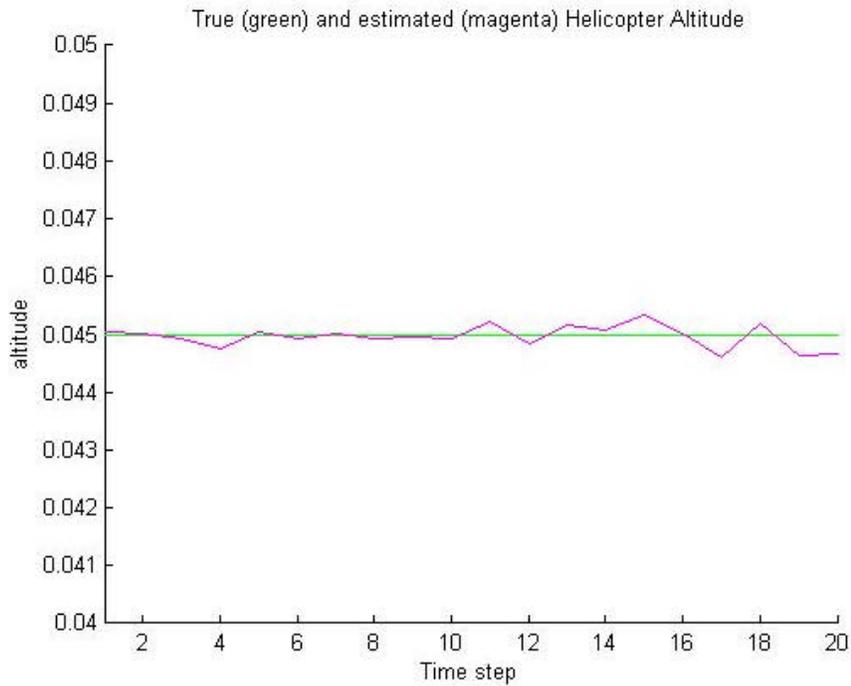


Figure 8.10: Actual and estimated altitude along the helicopter path

Figure 8.10 shows the actual and estimated helicopter altitude along its path (note the scale). As you can see, 3-D FastSLAM is capable of tracking the helicopter altitude fairly accurately.

8.6 Strengths and Weaknesses of FastSLAM

a) Strengths

- *Speed*: FastSLAM is fairly fast, particularly as compared with DP-SLAM.
- *Memory requirements*: FastSLAM remembers only the helicopter's and landmarks' positions. Thus, if the number of landmarks is not excessive, the memory requirement is low. Also, no additional memory is required to increase the resolution (as compared with grid-based methods like DP-SLAM).
- *Extension to 3-D*: FastSLAM extends easily to 3-D, with only incremental changes in computational requirements and time.

b) Weaknesses

- *Accuracy*: The accuracy obtained was slightly disappointing. Partly this is due to the large size of the landmarks and the tendency of large landmarks to appear to move along with the helicopter.
- *Robustness*: Two points make FastSLAM non-robust: 1) The data association problem is not accurate. Moreover, a failure in the data association has drastic effect on the Kalman Filter. 2) The shape of the obstacles is not taken into account, since the Kalman Filter only uses points. Thus, when a rock is seen from one viewpoint or another, the 'landmark's' position is moving as well, whereas a landmark should not move.
- *Incomplete map*: Unlike in DP-SLAM, the algorithm does not produce an actual map of the landscape, only a map of the landmarks. Some post-processing (such as appending scans to the final calculated poses) has to be performed before getting a map of the actual landscape.

9 DP-SLAM Results

9.1 Final Map

We were generally pleased with the results produced by DP-SLAM although we were somewhat hampered by the processing and memory constraints (see discussion below). Figure 9.1 shows the final map produced by DP-SLAM with the actual landmark positions and estimated and actual helicopter paths. DP-SLAM did not suffer from the difficulties that FastSLAM encountered related to landmark size. With DP-SLAM, large landmarks simply fill more squares and therefore do not appear to move as the helicopter moves around them.

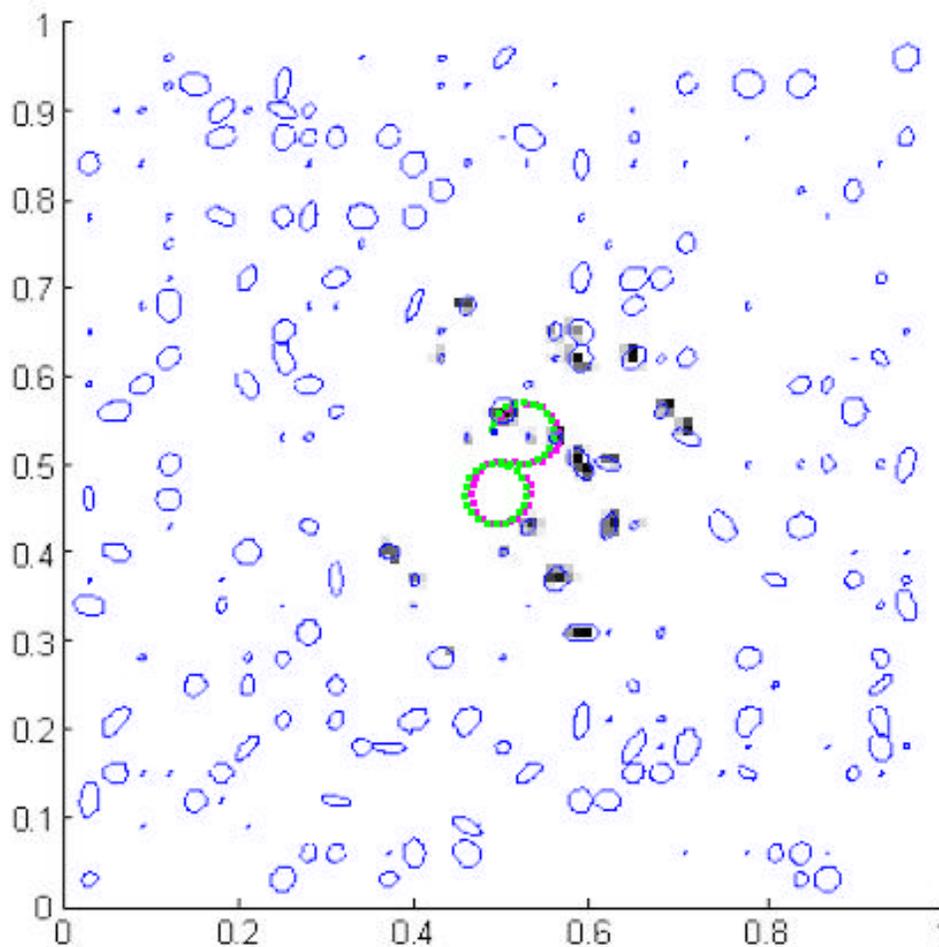


Figure 9.1: Final occupancy map produced by DP-SLAM overlaid with outlines (blue) of the actual obstacles in the environment. The shade of gray represents $P_c(x, \mathbf{r})$ for each square using the width of a square for x . Darker shades represent a higher probability of stopping the laser. The actual path of the helicopter is shown in green and the estimated path of the helicopter is shown in magenta. This simulation used 10 particles and was executed for 40 time steps.

9.2 Helicopter path

The estimated path of the helicopter tracked the actual path fairly well, even with a very small number of particles. From Figure 9.2, you can see that the algorithm does drift from the actual path when exploring new territory but does a good job of returning to the actual path when it is able to observe areas which it has seen previously.

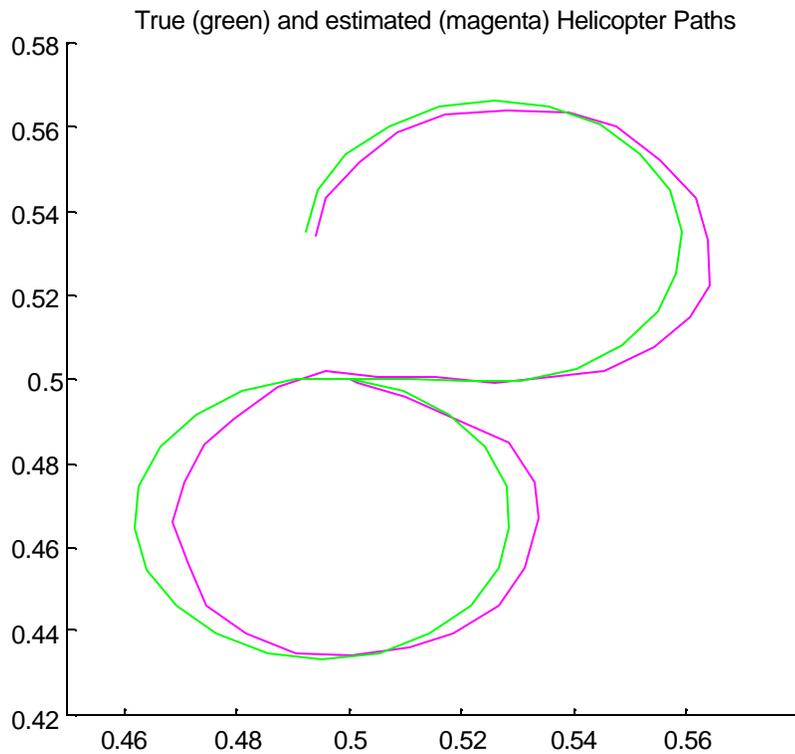


Figure 9.2: Actual and estimated helicopter path detail from the same simulation as above. The actual path is shown in green while the estimated path is shown in magenta. The path begins at (0.5,0.5) and starts out going clockwise the path ends near (0.5,0.53).

9.3 Performance

The one area where we were disappointed with DP-SLAM was the performance. Although this was partly due to the inefficiency of implementing tree data structures in Matlab, it is still clear that DP-SLAM is far more computationally intensive and uses much more memory than FastSLAM. This is largely due to the need to access and

update all the grid squares encountered by the laser casts rather than just operating on a few landmark positions.

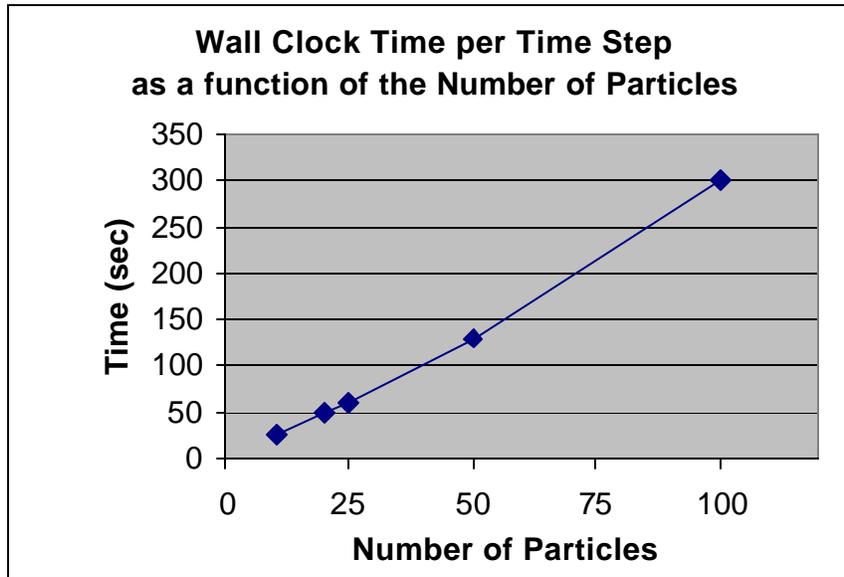


Figure 9.3: Average wall clock time required for each time step with varying numbers of particles. A more efficient implementation should be able to cut these numbers in half.

Because of the enormous memory demands of our implementation, the length of simulations was severely limited. With 100 particles, we were only able to execute 2 time steps before a machine with 1 GB of RAM exhausted its memory. Back of the envelope calculations indicate that the memory utilization is at least one order of magnitude higher than would be expected from the amount of data that needs to be stored. Presumably a C/C++ implementation of this algorithm would not suffer from the same problems. Since all simulations terminated by exhausting memory, we have not shown total memory usage. Instead, we give the amount of additional memory allocated per time step. This memory is almost all allocated during the map update phase indicating that the data being stored in the map is not stored efficiently.

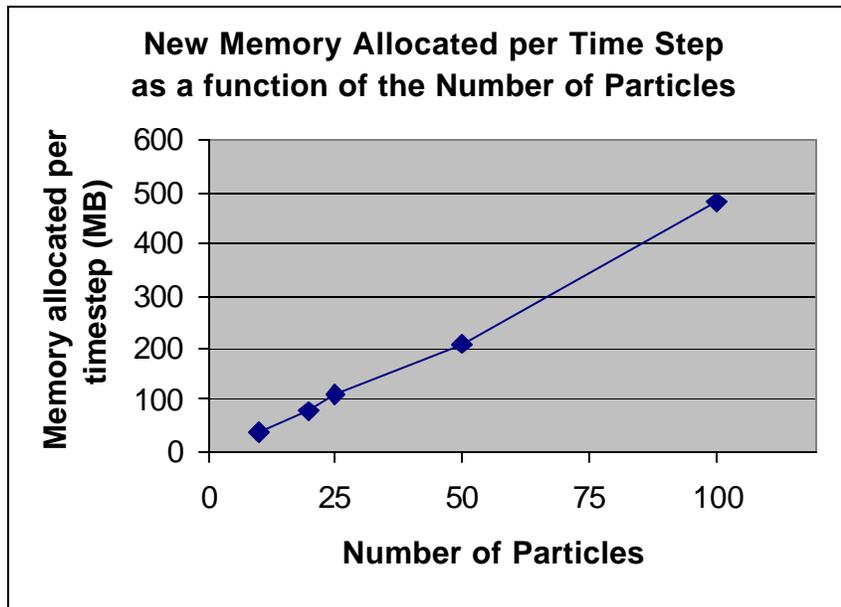


Figure 9.4: New memory allocated for each time step versus number of particles simulated.

9.4 3-D DP-SLAM

An extension of DP-SLAM to 3-D, which we did not implement, is a fair amount more difficult than extending FastSLAM to 3-D. The most straightforward extension to 3-D would involve using a 3-D occupancy grid instead of the standard 2-D occupancy grid, and using all of the 3-D rangefinder casts rather than processing it to a leveled 2-D scan. Creating a 3-D occupancy grid would, as with the 2-D case, result in automatic generation of maps rather than having to create a map during post-processing via appending scans to paths. In the 3-D case, that would mean being able to automatically map things like overpasses and tunnels, which could be very useful to a remote-controlled helicopter trying to fly autonomously.

However, instead of memory and time requirements being proportional to the area swept out by a 2-D laser rangefinder, they would be proportional to the volume swept out by the 3-D laser rangefinder. Given the prohibitive amount of time required to do DP-SLAM in 2-D within Matlab, we decided that trying to do 3-D DP-SLAM would be hopeless. We did find one paper describing using a 3-D occupancy grid (Kou, 2004), using stereo vision rather than laser rangefinders. While no mention of time or memory requirements was made, the results were nonetheless rather disappointing. We surmise that 3-D DP-SLAM may be impractical in terms of computational and storage needs.

10 Conclusions

From this project, we have learned about the main challenges of SLAM, as well as the strengths and weaknesses of FastSLAM and DP-SLAM. DP-SLAM was shown to provide a very accurate map of the world, even with a low number of particles. It is very robust, particularly when dealing with big objects, which are seen differently from different viewpoints. On the other hand, FastSLAM was less accurate but very fast, needed much less memory, and was easily extended to 3D. However, it requires a problematic pre-processing of the sensor information, since it treats the world as a set of point landmarks. Also, the shape of the landmarks cannot be taken into account, making the result even less accurate in chaotic worlds. In a very complicated world, when memory and speed are not main concerns, one should use DP-SLAM. On the other hand, FastSLAM provides a cheap algorithm in terms of memory and speed, but can have less accurate results. Finally, on the software side, we learned that one should never try to use pointers and trees in Matlab.

Bibliography

- Brenneke, Christian, Oliver Wulf, Bernardo Wagner. "Using 3D Laser Range Data for SLAM in Outdoor Environments," *IEEE/RSJ Conference on Intelligent Robots and Systems*, Oct. 2003.
- Eliazar, Austin and Ronald Parr. "DP-SLAM: Fast, Robust Simultaneous Localization and Mapping Without Predetermined Landmarks." IJAI, 2003.
- Eliazar, Austin and Ronald Parr. "DP-SLAM 2.0," ICRA, April 2004.
- Keren, Yaron. "Data Structures & Algorithms Toolbox." Available from Matlab Central (<http://www.mathworks.com/matlabcentral/>).
- Kou, Tatsunori, Kenji Suzuki, and Shuji Hashimoto. "Three-dimensional mapping utilizing stereo vision and Bayesian inference," *Proceedings of SPIE*, Vol. 5603, 2004.
- Montemerlo, Michael, Dirk Hahnel, David Ferguson, Rudolph Triebel, Wolfram Burgard, Scott Thayer, William Whittaker, and Sebastian Thrun. "A System for Three-Dimensional Robotic Mapping of Underground Mines." In *IEEE International Conference on Robotics and Automation (ICRA)*, 2003.
- Montemerlo, Thrun, Koller, Wegbreit, "FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem," *Proceedings of the AAAI National Conference on Artificial Intelligence*, 2002.
- Thrun, Sebastian, Mark Diel, Dirk Hahnel. "Scan Alignment and 3-D Surface Modeling with a Helicopter Platform," *The 4th International Conference on Field and Service Robotics*, 2003.
- Zolotykh, Nikolai Yu. "Matlab Pointer Library." Available from <http://www.uic.nnov.ru/~zny/matlab/pointers/pointers.html>.