Markov Chain Monte-Carlo Localization For Outdoor Robots

Robert MacLachlan

April 14, 2003

Introduction

This document describes software developed for feature-based position estimation in outdoor robots. There are three major parts: description of algorithms, performance evaluation and software interface documentation.

The MCL Framework

MCL is an extremely general framework for localization that can be used with almost any sort of sensor and map. Its power comes mainly from two aspects:

- The use of a probability model that reformulates the problem of global localization as a tractable local conditional probability. This allows position information to be gleaned from sensor inputs that at any given time provide only very vague constraints on our global position.
- The use of a weighted sample to represent the distribution of the position estimate. This gives an efficient sparse representation of an arbitrary probability distribution.

All knowledge about the sensor and map is embedded in a probability function P(***s* | *l***), the probability that we will see the current sensor input ***s*** given that are at a location ***l***. This question is easily answerable on a strictly local basis by simply comparing what we would see if we were at position ***l*** in the map with what we do see.

Each sample is a tuple <***l***, *w*> of a location ***l*** and a weight *w* (0..1) . ***l*** should be thought of as a location hypothesis, with *w* a representation of the degree of support for that hypothesis. The sample set is kept normalized with the sum of all weights 1, so it resembles a probability density function.

The Core MCL Algorithm

On each iteration, the inputs to the algorithm are the relative motion since the last update $\boldsymbol{m}$ and the current sensor input $\boldsymbol{s}$. This is pseudocode for one update cycle:

```
    For each sample $<\boldsymbol{l}_0, w_0>$, update it to $<\boldsymbol{l}_1, w_1>$, where $\boldsymbol{l}_1 = \boldsymbol{l}_0 + \boldsymbol{m} + m\_dither$ and
    $w_1 = w_0 * P(\boldsymbol{s} \mid \boldsymbol{l}_1)$

    Normalize the sum of all sample weights to 1.

    For each sample whose weight falls below a threshold $min\_weight$:
        If the number of samples exceeds $samples\_goal$, then delete the sample
        Otherwise generate a new location for the sample:
            Choose another sample $<\boldsymbol{l}', w'>$ at random with probability
            proportional to $w'$.

            The new sample is $<\boldsymbol{l}_1, w_1>$, where $\boldsymbol{l}_1 = \boldsymbol{l}' + r\_dither$ and
            $w_1 = w' * P(\boldsymbol{s} \mid \boldsymbol{l}_1) * penalty$

    Normalize the sum of all sample weights to 1.
```

The algorithm has two major steps: motion update and resampling. Motion update changes the position of the distribution to reflect the known motion since the last update and updates the sample weights to incorporate the new sensor input. Resampling adapts the distribution of samples to have more samples in high probability areas.

Motion update basically adds the relative motion to each sample location. An additional random offset *m_dither* is added to represent the increase in uncertainty due to error in the relative motion estimate. This is a Gaussian random with standard deviation set to model the motion error. With an INS, this term is small, requiring additional dither to be introduced in the resampling process. The motion update step also incorporates the new sensor input into the sample weights by multiplication with the conditional probability.

Resampling is done on samples with very low weights, currently 1e-5. These samples contribute little to the shape of the distribution but still take work to update. If there are many samples (> 200), then low weight samples are simply deleted from the set. This allows us to start out with a large sample to represent the initial position uncertainty, and then to reduce the sample to a manageable size as information is gained. Once the sample is pared down it is maintained at a fixed size by replacing low-weight samples with higher weight ones. Note that the process of "resampling" in a new location does not retain any information from the old sample – it is purely a matter of reusing memory.

The new location is chosen based on the location of another randomly chosen sample. The sample is drawn by rejection sampling so that the probability of drawing a sample is proportional to its weight. This approximates a draw from the underlying distribution represented by the weighted sample. The new sample is at this location plus a random offset. The random offset is a Gaussian with standard deviation chosen so that its peak value is the drawn weight *w'*. This models the contribution of a single sample to the overall distribution as a Gaussian with mean at the sample location. The purpose of adding dither to the resampled location is to give samples usefully distinct locations.

In resampling, the weight of the sample is computed as though *w'* were the previous weight, but with an additional *penalty* factor (0.01). This penalty prevents new samples from significantly affecting the distribution until they prove themselves by gaining weight over a number of iterations.

Convergence Detection and Failure Recovery

After each MCL iteration we use this procedure to evaluate the result distribution:

> Compute the weighted mean and covariance of the sample.
>
> If sqrt(COV_{xx} + COV_{yy}) is above 4 meters, then sample has not collapsed, and no further action is required.
>
> If the sample has collapsed, then the mean is a potential position fix ***l*.**
> Compute the probability of ***l*** given the last *k* sensor inputs**:** $P(\boldsymbol{l} \mid \boldsymbol{s}_{n-k} .. \boldsymbol{s}_n)$
> If this is less than 0.5, reinitialize the sample and restart localization.
> Also restart if the mismatch between the fix and the prior absolute position exceeds six sigma.
>
> If the number of resamples on the last iteration is < 15, then the fix is considered valid. If this test fails, we don't reset; we just wait for the resampling to settle down before reporting a valid fix.

If MCL succeeds, then the sample will become concentrated in a small area. This can easily be detected by computing the weighted covariance of the sample and comparing it to a threshold. However, MCL may also fail by falsely localizing or become lost by spuriously rejecting the correct position without localizing. Failure is particularly likely in the case of excessive unmodeled error in the map/sensor or relative motion. For example, in the second segment of the test data, the robot drives on a road that is not in the map. In the presence of inconsistent input, MCL will converge on the most probable fix, even if that fix is highly improbable in absolute terms.

To guard against false localization we compute historic probability P($\boldsymbol{l}$ | $\boldsymbol{s}_{n-k}$ .. $\boldsymbol{s}_n$). It is difficult to compute such a probability during localization, but once we have a tentative fix it is straightforward to validate it retrospectively. During localization we log sensor and relative position data in a ring buffer. Once we have a tentative fix, we can use it to superimpose the recorded path onto the map and check for match quality. The probability is computed as the geometric mean of P($\boldsymbol{s}_j$ | $\boldsymbol{l}_j$) for j from n-k to k. This is the same sensor/map probability function used in the localization step, so no additional sensor or world model is required.

When computing the historic probability, we take into account the uncertainty of past positions due to cumulative error in the relative motion estimate. This reduces the weight of older observations, although the effect is small if the relative motion error is small.

The historic probability test does not guarantee a correct fix, but has not generated any false fixes in testing so far. We also check that the fix is consistent with the prior position estimate, which at least bounds how wrong we can be.

Since we only reset the distribution when it has collapsed, it is possible that we could fail to localize indefinitely. Some additional tests should be developed to handle this case. An advantage of resetting only when the sample collapses is that it is conservative in that it won't reset when the sample still covers the correct location. Also, it tends to reset at a "good" time. For example, if we are driving on a road not in the map, then there is basically no support for our position. Resampling proceeds at a furious rate and prevents the distribution from collapsing. When we return to a mapped road, the distribution rapidly collapses on the road. Since the recent probability is very low, it will reset at that time.

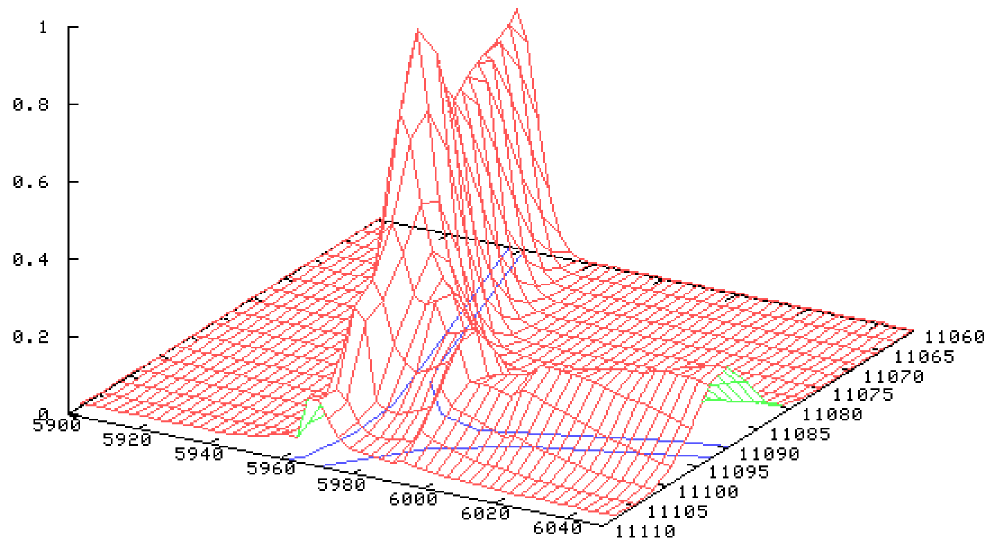
We assume that some other navigation system will update our last valid fix for known relative motion in the event that we start reporting a localization failure. It is thus desirable to revoke our valid flag whenever we suspect that accuracy may be degraded. A high resampling rate is suspicious because it indicates that the distribution is changing rapidly, however this is not necessarily a bad thing. Resampling also spikes when we acquire a useful new constraint, so bursts of resampling also precede correct localization.

The Sensor Model

Although the implementation of the basic MCL has an abstract interface that supports fairly arbitrary sensors and maps, the only implemented sensor/map combination is based on linear features such as roads. A map is a set of features represented as sequences of two-dimensional points.   The sensor returns a 2D ray in robot coordinates, represented as an offset and a direction.

P(***s* | *l***) is computed by transforming the ray into absolute map coordinates using the location hypothesis ***l,*** then finding the point on a mapped feature that is the best match for the ray ***s.*** This point must be the closest point on some line segment in the sensor’s error radius. It is not necessarily the closest point on the closest segment because the orientation match is also considered. The translation and rotation error are modeled as independent Gaussian distributions with variances that are specified by the sensor.

Here is an example of probability vs. position for a particular road direction that is parallel to the road going toward the back of the picture. Note how the direction match significantly influences the probability. The road margins are shown in blue:



In principle it is possible to infer position from negative observations: we don't see a road, so we are not near a road. We don't do this for several practical reasons. One is that to determine that something should have been seen when we are in a given location requires an accurate model of the sensor's field of view, range, etc., which is not otherwise needed. Also it seems plausible that false negatives will be more common than false positives, rendering this data weaker. Furthermore, the negative data is already quite weak. Knowing we are not on a road does not narrow down our location nearly as much as knowing that we are.

Note that as far as map-based localization is concerned, map error and sensor error are indistinguishable. We arbitrarily attribute all of the error to the sensor. Systematic errors such as missing roads in the map are not well modeled as Gaussian and white, and do cause problems for MCL. Actual linear feature sensors will also likely contain some sort of internal tracking filter that causes a low-pass cutoff in the output. We attempt to mitigate this problem by discarding sensor readings taken at positions that fall within one sigma of the position that the last used value was taken at. This is rather ad-hoc, but should serve to increase the independence between the measurements.

The sensor model is also responsible for initializing the distribution. What we do is wait for a feature to be observed, then initialize the distribution to be the points in the map corresponding to all features within the six sigma uncertainty of our prior absolute position. The points are placed at a uniform spacing along the features and laterally dithered in a fixed Gaussian distribution. If the uncertainty radius is too large or there are too many features, then too many initial samples would be required. In this case, we defer initialization until the uncertainty is small enough. Sample sizes typically range from 200 to 2000.

Previous Work

The published algorithm for Markov Chain Monte-Carlo localization (MCL) [FOX99] was used as a starting point for algorithm development. These are the major changes we have made:

- Resampling is done only on samples from low weights. In effect, when the sample has a good fit, then the algorithm degenerates to a sparse Non-Monte-Carlo Markov localization algorithm. The main reason for this change was to reduce spurious condensation (loss of entropy) in the sample during periods where the sensor is not contributing any useful constraints. Information in the sample is preserved better, especially in the case where the probability of a sample drops briefly due to some glitch in the map or sensor. Somewhat counterintuitively, this improved convergence in our testing because nearly correct initial samples weren't thrown out during episodes of spurious condensation. This will tend to limit the resolution of the estimate, since no new samples are introduced as long as all samples are moderately probable.
- The distribution is initialized to be near features in the map, rather than uniform. This is attractive partly because we have an accurate absolute heading.
- Dither is added when resampling. Without this change samples stack up in basically the same place when the relative motion error is small (as it is with an INS.) In order to broaden the search it is necessary for entropy to be added somewhere.
- In resampling, the prior weight of the new sample is a fraction of the base sample weight, not 1/sample-size. The prior weight has little effect as long as it is reasonably small, but it seems reasonable to factor in the weight of the base sample. With the current parameters this typically gives a weight about 2x larger.
- We use simple rejection sampling for resampling. Because the rate of resampling is greatly reduced, there is little efficiency penalty.
- The adaptive sample size algorithm based on pre-normalized weights has been replaced with a fixed sample size goal. There is little benefit to reducing the number of samples below a certain level, and it risks losing sample diversity that may prove useful later.
- We do not add random uniform samples. Cases where the robot becomes lost are detected from the historic probability and localization is restarted.

Mixture MCL [THRUN01] addresses some of the robustness problems with MCL by introducing a dual probability function $P(L \mid S)$. Unfortunately, this function is an approximation of what we are trying to estimate in the first place, and cannot be computed locally on a map. KD trees are used to build an inverse map, at the cost of considerable space and implementation complexity. Our use of the historic probability in the convergence test is conceptually similar, and gains many of the same robustness benefits for relatively little cost. We don't realize the time benefits of smaller samples, but the runtimes we get are quite reasonable, and we avoid the daunting overhead of building an inverse map of a large outdoor environment.

Differences in Requirements

Much previous work was done on indoor robots with only odometry. With no absolute heading reference, heading needs to be estimated as well as position. The XUV has a quality INS, so our relative motion estimate is much better, and we have an absolute heading reference. From the INS specifications we determined that there was no need to estimate heading, which significantly reduces the size of the search space. More subtly, the greatly reduced relative motion error eliminated some dithering, requiring algorithmic changes.

Another major difference is that we use a much higher-level representation of the feature sensor input. In the indoor work, a feature is often simply the presence or absence of a detectable object using a range sensor (ultrasound or lidar.) In our application we are given a map of linear features (e.g. roads) and output from a sensor that can detect that type of feature. This makes our life easier because we have less data to handle, and also because it allows us to further reduce the size of our search space from 2D to the fractal dimension of the road system.

Unfortunately, the external feature extraction process will also inevitably introduce error. We were not able to evaluate this since we did not have any feature detector available. However, the tolerance for apparent map errors observed in testing suggests that some robustness in the face of sensor error can also be expected.

References

[FOX99] Monte Carlo Localization: Efficient Position Estimation for Mobile Robots
D. Fox, W. Burgard, F. Dellaert, and S. Thrun
Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)., July, 1999.

[THRUN01] Robust Monte Carlo Localization for Mobile Robots
S. Thrun, D. Fox, W. Burgard, and F. Dellaert
Artificial Intelligence Journal, 2001

Performance Evaluation

The Simulation

Performance was evaluated using a prepared map and a trace of robot positions collected while driving on roads. Since there was no road sensor, sensor input was simulated by reporting that a road was seen at the current position oriented in the direction of travel.
This simulation has a number of limitations:

- There is no possibility of localizing to accuracy of much better than the road width, since there is no indication of our position on the width of the road.
- Direction of travel is only a rough indication of the road direction, limiting the accuracy of the "measured" orientation.

The path data was divided into three contiguous parts, the second of which is mainly driving on a road not in the map.

The Map

The supplied map also had a number of flaws. A serious but fixable flaw is that the map seems to have been prepared more for visual appearance than automated interpretation. The map is a collection of lines which "looks like" a road map. In keeping with looking like a map, both margins of the road are drawn. However there is no established relation between the features for one side of the road and the other. In fact, there is no consistent relation. Often different features represent the two margins. Sometimes, especially on dead-end roads, one feature is used for both margins. The situation is particularly confused at intersections. Often when a side road branches off, the exit appears blocked by the margin of the main road.

Such a map would be useless for tasks such as following a road. Fortunately the map flaws serve mainly as an excellent demonstration of how MCL can deal with ambiguous and even inconsistent data. All that we really need to know is how "roadish" a given piece of map looks, and what the orientation of a road would be.

In order to best use the map of road margins, we adapted the probability function to sum the probability for all features in the vacinity. This creates a probability distribution with a single peak in the center of the road, instead of a bimodal one where driving 2m off the road is considered just as likely as driving in the middle of the road.

The other map flaws are more unavoidable, though they could perhaps be reduced. Basically, it contains errors. In one case, the robot drives on a road not in the map. In some other cases, a degree of mismatch between the path and map develops that is suggestive of excess map error, excess relative motion error, or both. Or maybe the robot was just driving on the shoulder.

Position Error Model

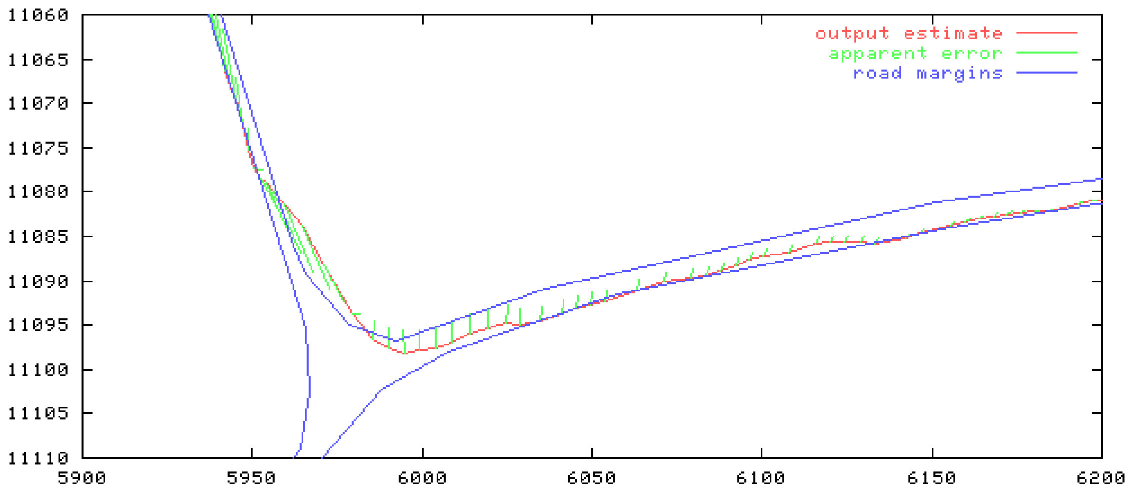
The simulation results shown here were done using a 45m six sigma bound on prior absolute position. Uniform random five sigma offsets were added into the absolute position reported in the trace. This error is in addition to whatever absolute error was actually present in the initial data. Note that due to the way the distribution is initialized, the component of position error normal to the road has little influence on performance (this is a real effect, not a testing artifact.) The simulated position error is fixed for the duration of the run, though varied values were used (see below.) Some testing was done with 300m uncertainty also.

Position Estimate Quality

The only position ground truth data available is the input absolute position less the simulated position error. We can’t with any confidence evaluate our ability to improve on the fairly high quality input estimate, but we can simulate what performance could be seen with a worse initial position estimate. We refer to the difference between the uncorrupted input position and the output estimate as the *apparent error*.

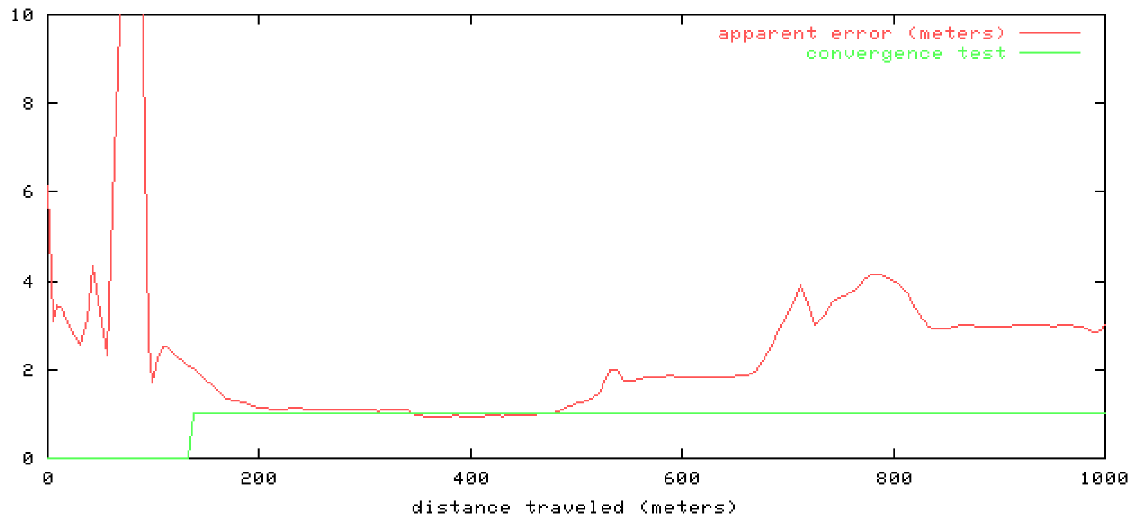
Data Set One

Here is a plot of the point of successful localization during a typical run with the first data set:


output estimate
apparent error
road margins
11060
11065
11070
11075
11080
11085
11090
11095
11100
11105
11110
5900
5950
6000
6050
6100
6150
6200

The robot proceeds from left to right, and localizes after turning the corner. The green error vectors are drawn from each output position estimate to the corresponding uncorrupted input position. There is 6x vertical exaggeration to make the lateral position error clearer. Note that according to the input position, the robot appears to drive well off of the road before the turn.

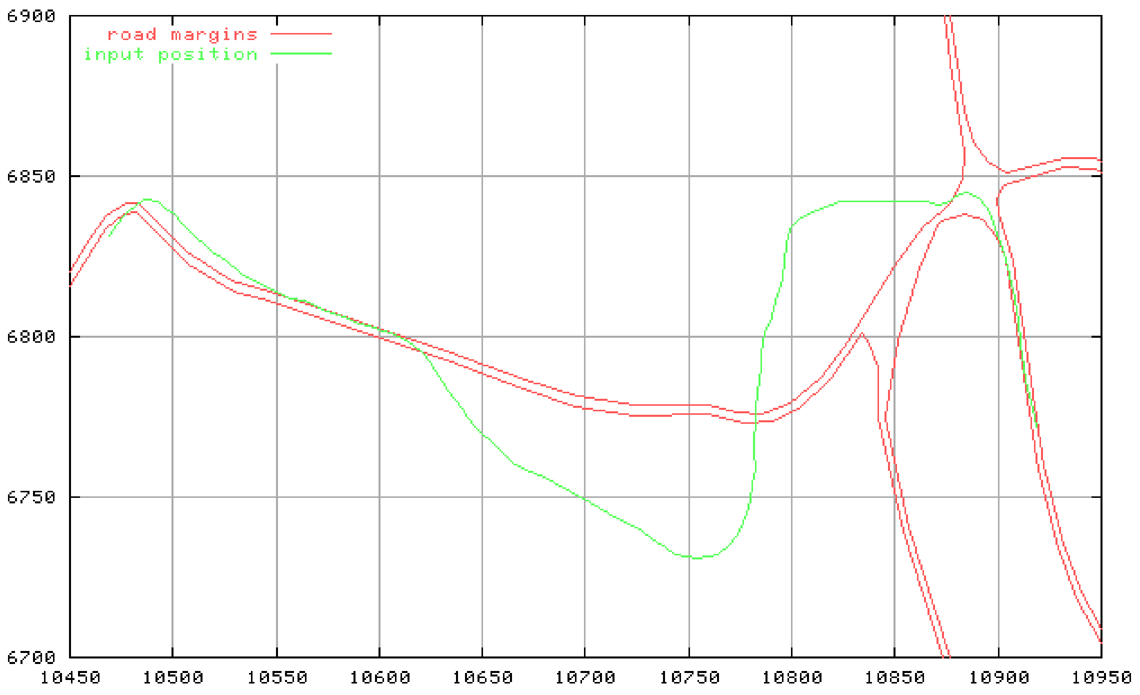
In the first data set, the apparent errors seen are around the width of the road or less (~4m), which is the best that can be expected without a road position sensor.   Here is a plot of apparent error and the convergence test vs. distance traveled:



The apparent error increases by about 2m over 1 kilometer. Some of this apparent error is probably due to cumulative error in the input estimate. This is supported by an observed deterioration in the third data set of the matching between the input estimate and the mapped roads. The steps in the apparent error correspond to bends in the road that provide useful position constraints to MCL. Note that the estimate is indeed quite accurate by the time that the convergence test is satisfied.

Data Set Two

During most of data set two, the robot is driving on a road that is not in the map:

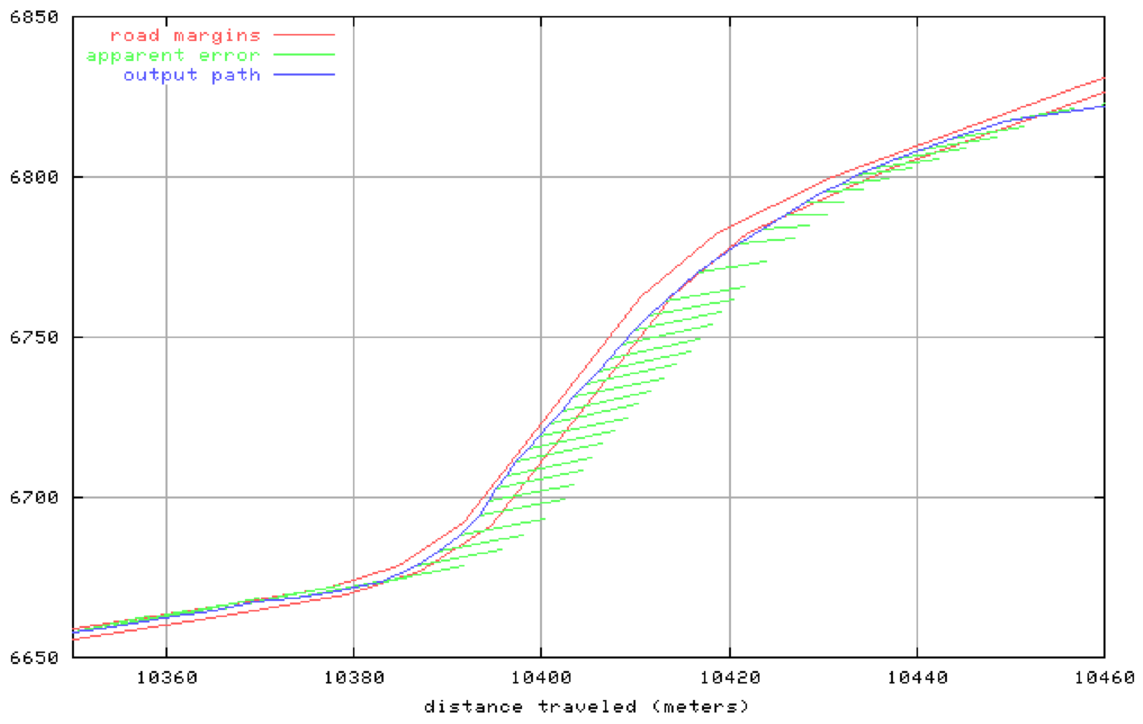


The algorithm does not find any fix during this run, which is the best that can be expected. The algorithm does reset several times due to the distribution collapsing at a position with low probability.

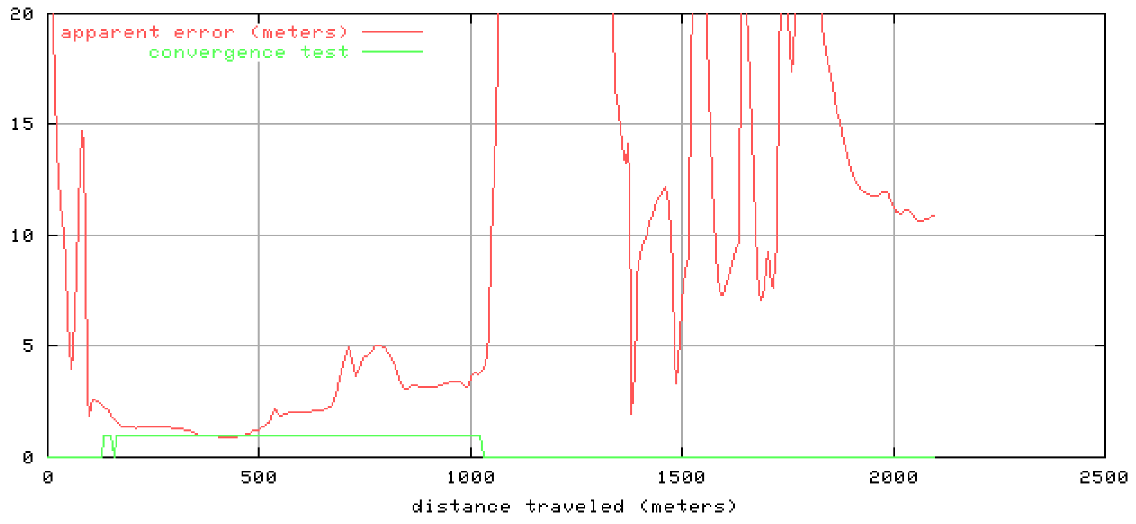
Note that even after the path returns to a mapped road, a considerable offset to the east is visible. This error persists into the third data set.

Data Set Three

In the third data set, the position does converge, but with a 10m apparent error. This error is due to a clear mismatch between the input position and the map:



It is impossible to tell for sure whether this is a map error or an input position error, though there is some reason to suspect a position error. One comes from the results of running with all three data sets concatenated:

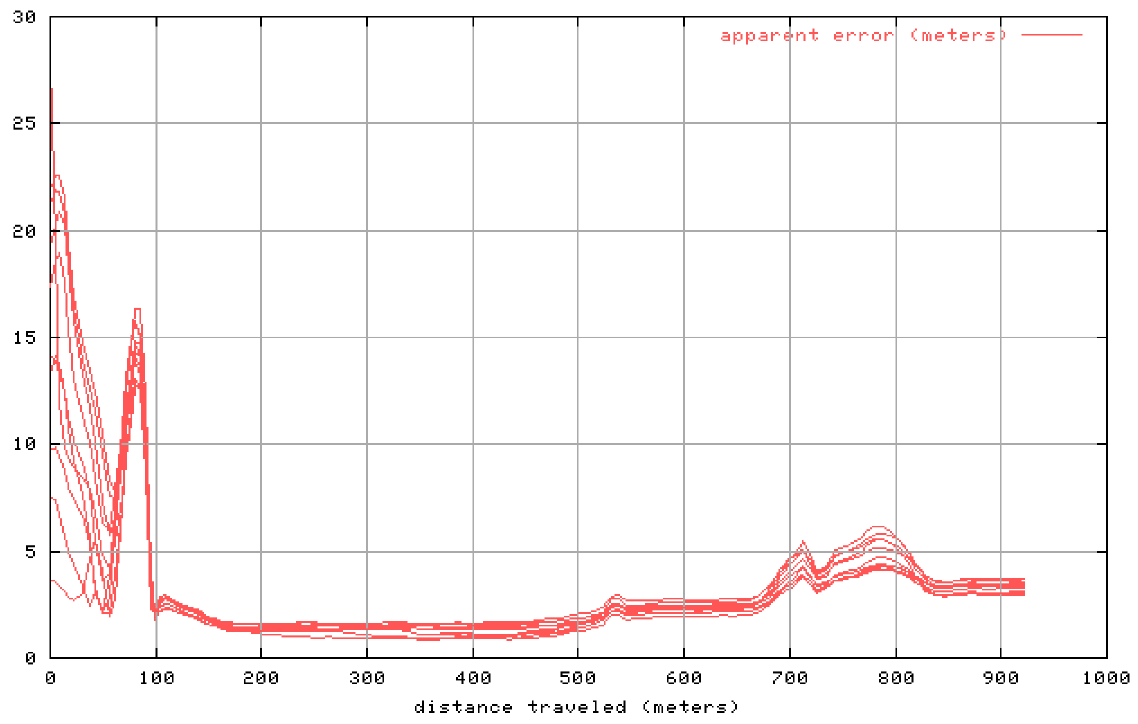


If you ignore the spiking between 1km and 1.8km due to driving off the map, there is something that looks like a linear increase in apparent error with distance traveled. Looking at the input path superimposed on the map is also suggestive of a cumulative drift to the east. The 10m displacement at the end holds across several bends in the road which suggests some sort of systematic error, and not just an error in hand digitization of the road.

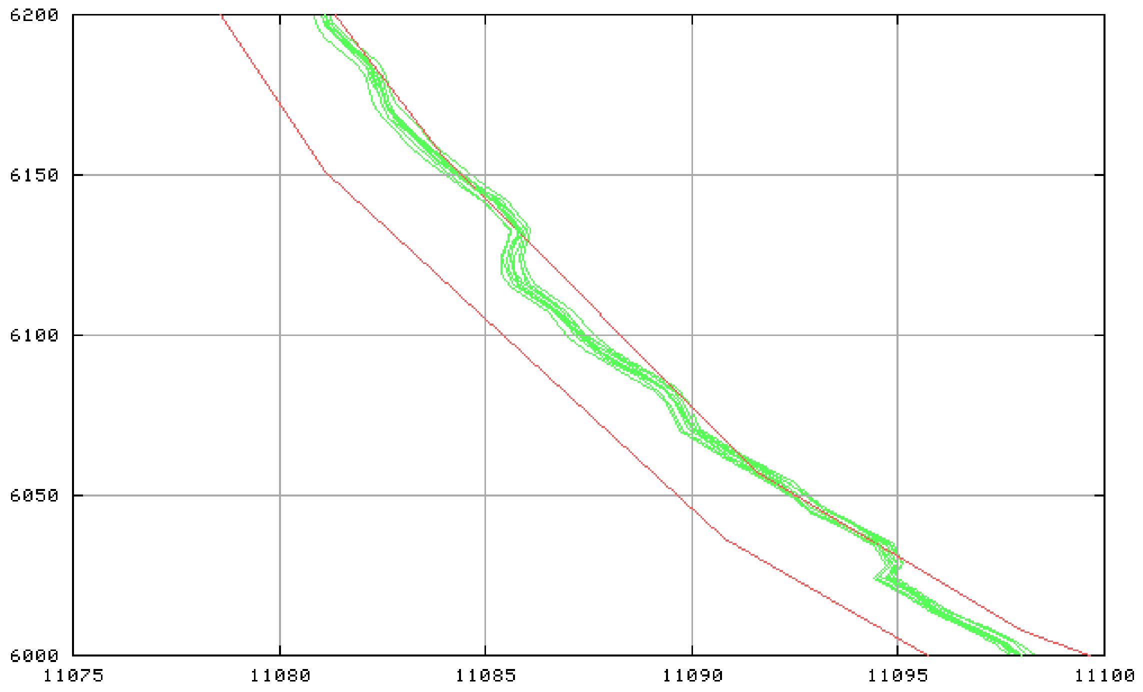
Note that when the entire concatenated data set is run, the convergence test has not been satisfied by the end of the third part, whereas it does converge if the third part is run by itself. This is because we start the third part with a messier distribution than we would have if we had reinitialized at that point. Graphs of the internal convergence factors suggest that convergence was well underway when the data ended, and the non-validated output position is basically the same as the one that we get when the third part is run alone.

Result Stability

Since MCL is a probabilistic algorithm using random numbers, one important question is the stability of the result. We evaluated the stability by doing multiple runs with different random seeds and different simulated absolute position error. Here is a graph of the apparent error for ten runs of the first data set:



During this test the initial position estimate varies over tens of meters, whereas the output estimate varies less than three meters. This is a plot of a portion of the output estimate path during the same ten runs:



Memory Usage

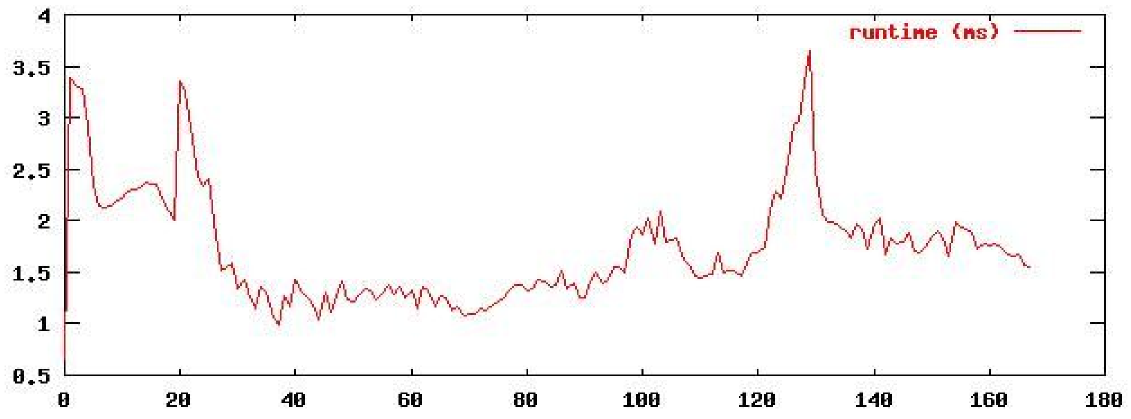
The map dominates memory usage. The original map data was 6 megabytes. We preprocessed the map data to remove redundant vertices and subtract off a fixed origin. We deleted vertices that could be removed without introducing more than 0.5m error, which reduced the number of vertices to about 1/7. Subtracting the offset allows points to be represented with a single float to sufficient resolution, approximately halving in-core size.

The size of the in-core map data is 850kb. Of this, about 75% is indexing data that allows the features in a given area to be rapidly computed.

CPU Usage

Performance was evaluated on a 1.4GHz Athlon processor under Linux. Elapsed time for each iteration was measured to sub-nanosecond resolution using the rdtsc instruction. Due to interrupt handling, etc, reproducibility is more like +/- 20ns. The program was run at high priority with sleep calls between iterations to avoid preemption during an iteration. Since performance does not improve as fast as clock rate, it is conservative to extrapolate to a slower clock by taking the clock speed ratio. Architectural differences are more difficult to predict.

Here is a typical run:



The average elapsed time per iteration is about 2ms, but there is considerable variation depending on the amount of resampling and the local complexity in the map.

Scaling

One of the problem size parameters is the absolute position uncertainty. Large uncertainties increase runtime and also slow convergence due to increased ambiguity. In a run with position uncertainty of 300m, 7500 samples were used. Convergence was still good, but runtime is proportional to sample size, so worst case runtime on iterations before convergence was about 20x greater. Total CPU to localize was still less than one second. It appears that the scaling of sample size with radius is about $r^{1.6}$. With sample sizes in this range the memory usage of the sample itself is still small compared to the map data.

The other size parameter is the map size. Map data size scaling should be the same as sample size. In both cases, the fractal dimension of the road network (~1.6) determines the scaling. For maps larger than 50x50 km or so, it would probably be worth looking into indexing structures with more than two levels.

Summary

Overall, position performance seems as good as can be expected given the limitations of the input, and speed/space efficiency seems quite acceptable. The main limitations of this approach seem not to come from the algorithm itself, but from the accuracy of the map and the road sensor.   The biggest weakness of any map-based navigation scheme is the requirement for an accurate map. Significantly, the MCL variant described seems to provide a good degree of robustness against missing roads in the map. If the map is wrong, we can't figure out where we are, but we do have a good chance of realizing we are lost because the shape of the road does not match the shape of our path.