Advanced Localization

Carnegie Mellon University

October 27, 2003

Scott Lenser
Manuela Veloso

Particle Filters

- We have seen how to use particle filters to track the robot's position over time.
- The particle filter technique was first applied to object tracking and is known as Blake's Condensation in this context.
- The particle filter technique applied to localization is commonly called Monte Carlo Localization and was introduced by Dieter Fox.

Particle Filters - Commentary

- We now have a solution to the localization problem.
- We can track a probability density reflecting the robot's position in the environment.
- We can update this probability density for robot motions.
- We can update this probability density for sensor readings.
- We can compute position estimates and other summary statistics from the probability density.

Localization Subproblems

- Tracking — the process of tracking the robot's position from a known starting position
- Global localization — the process of finding the robot's location from an initially unknown position
- Kidnapped robot problem — the problem of localizing a robot which was localized but has been moved without its knowledge to another location

Particle Filters - Possible Problems

- Our solution is only as good as our assumptions and approximations.
- If we have insufficient samples, we can fail to converge, converge to an incorrect result, or introduce bias into our estimates.
- If our motion model has the wrong mean, the position estimate will consistently be biased away from the correct robot position.
- If our sensor model has the wrong mean, the position estimate will be biased away from the correct robot position on every sensor reading.
- If either model is missing events that could happen, the localization could fail if one of these events occurs.
- Between approximation errors and model errors, it can be very difficult to handle rare events while using a reasonable amount of processor time.

Particle Filters - Possible Problems

- If our motion model has too small a deviation, we won't use the sensor readings enough.
- If our motion model has too large a deviation, we will rely on the sensor readings too much.
- If the sensor model has the wrong deviation, we get similar effects to the motion model deviation being incorrect.
- If any of the deviations is wrong, the uncertainty in the resulting position will also be wrong.

Particle Filters - Possible Problems

- We assumed that sensor readings are independent given the robot's pose.
- If we have any unmodelled bias in our sensor readings, this assumption will be violated.
- This can make us overestimate our confidence in our position estimates.
- Common ways to mitigate this effect are to reduce the amount of information used from each sensor reading or only use sensor readings that are separated by a minimum time/distance interval.

Particle Filter Sample Counts

- It is desirable to use a low number of samples because the computation time required for localization is $O(n)$ where n is the number of samples.
- But accuracy of the approximation decreases with sample count size.
- The number of samples needed to accurately approximate the pose probability density depends on the shape of the probability density.
- Probability densities which have larger high likelihood areas require more samples to adequate represent the different possibilities.
- Many more samples are required for global localization than for tracking.

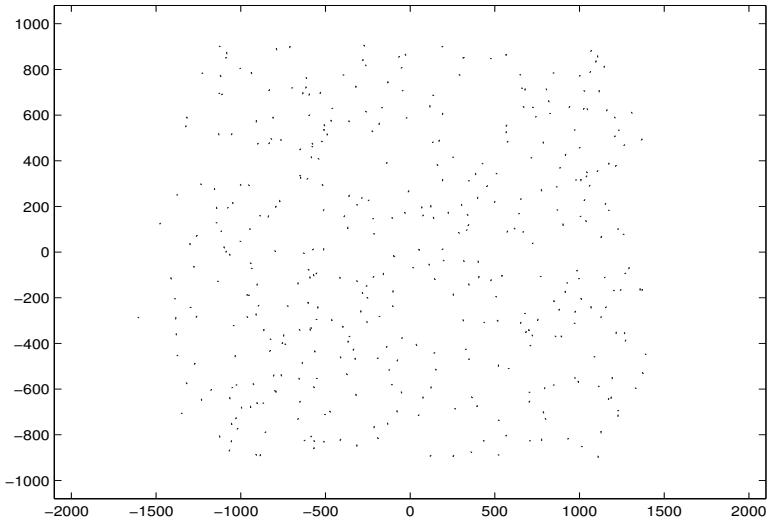
Particle Filter Sample Counts

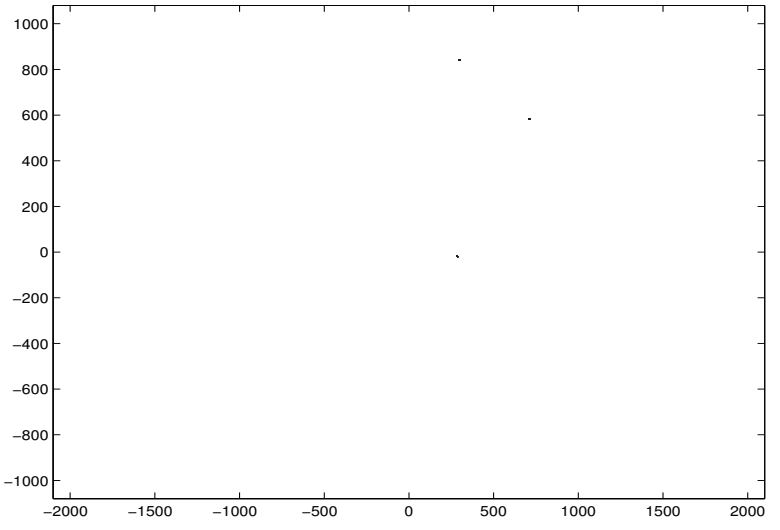
- Too many samples results in excessive CPU usage.
- Increasing the number of samples increases the accuracy of the localization.
- But, errors in the models always represent a significant contribution to the total error of localization.
- Errors in the models cannot be compensated for by adding samples.

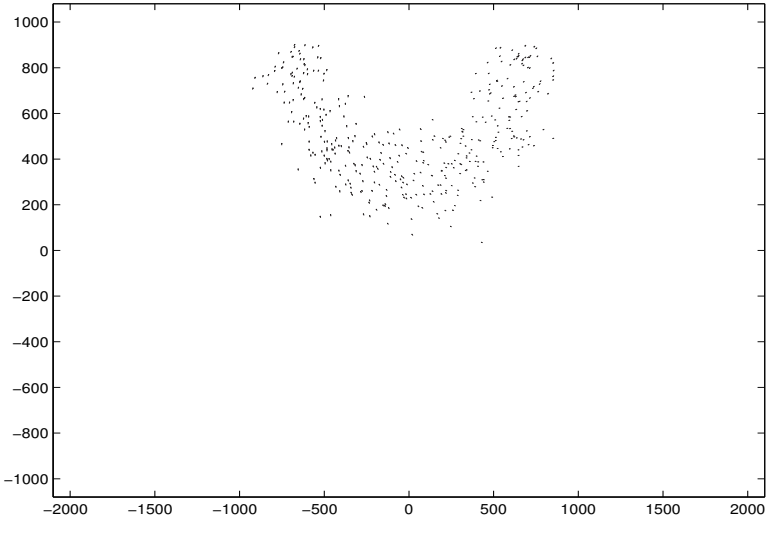
Sample Counts and Global Localization

- Global localization requires a large number of samples because the pose probability is so diffuse when the robot is lost.
- Global localization can easily require 2 orders of magnitude more samples than tracking.
- If the sample count is too low during global localization, only a few samples will be consistent with any particular sensor reading.
- This will cause the probability density to prematurely converge to a usually incorrect solution.

Sample Counts and Global Localization







The Quest for Autonomy

A short summary of failure methods for localization

- Ommissions in motion model, such as:
  - Biased movement estimates
  - Collisions
  - Movement of robot by humans
  - Slippage of robot on floor
  - Effects of partially drained battery
- Ommissions in sensor model, such as:
  - Biased sensor estimates
  - Effects of changes in environment (lighting conditions, etc.)
  - Failure/degradation of sensors
  - Interference by humans
- Correlated (non-independent) movement estimates, sensor estimates
- Insufficient number of samples

The Quest for Autonomy

- Obviously, localization can fail.
- But we would like robots that are able to operate without human intervention while maintaining their position over long periods of time.
- We could try to cover all of the possible failure cases in the algorithm.
- But it is easier to detect failure and go into a failure recovery mode instead.
- This motivates the need for failure recovery.

Failure Detection

- In order to recover from failures, we first need to detect them.
- We need a way to distinguish between a failure of the localization and its normal performance.
- When localization is working well, most sensor readings mostly just confirm the robot's current estimate of its position.
- A measure of how well the pose probability density predicts sensor readings therefore acts as a good tool for measuring localization performance.

Failure Detection

- The likelihood of the sensor reading given our current belief is $\int_{l^t} P(o^t|L^t)B_-(L^t)$.

- If we look back at our sensor update equation,

$$B(L^k) \propto P(o^k|L^k)B_-(L^k)$$

  we see that the value we want is simply the sum of the unnormalized sample weights after the sensor update but before the renormalization step.

- This performance measure will go down as the sensor readings become more unlikely and go up as the sensor readings become more likely which is what we want.

Failure Recovery Method

- We need to a way to recover when our performance measure says things are going badly.
- An easy way to recover is simply to reset the localization to saying the robot is lost.
- We might as well use our latest sensor reading as well since it is the one that told us we were mistaken.

Failure Recovery Method

- To do this, we need to generate samples distributed according to

$$P(L^t|o^t)$$

- This is difficult but feasible to do in general
- As long as the resulting samples assign reasonably high probability to the robots actual location (compared to other locations) this will work ok.
- This allows us to get away with fairly poor approximations of the desired distribution.

Failure Recovery Rule

- We now have a performance measure for detecting localization failure.
- We also have a method for failure recovery
- We now need a rule for deciding when to invoke our failure recovery based on our perfomance measure
- We would like the rule to gracefully handle cases where our performance measure is ambiguous.

Failure Recovery Rule

- Idea is to approximate the probability that the localization has failed.
- Let $p_s = \int_{l^t} P(o^t|L^t)B_-(L^t)$.
- We will use a simple approximation

$$P(F^t = 0) = \begin{cases} 1, & \text{if } p_s > p_t \\ p_s/p_t, & \text{otherwise} \end{cases}$$

- F^t is 1 if the localization has failed at time t and 0 otherwise
- p_t is a threshold probability at which we start to believe the probability that the localization has failed is non-zero.

Failure Recovery Rule

- We can now use the probability of failure in the sensor update rule.
- Recall, the basic sensor update rule is:

$$B(L^k) = \eta P(o^k|L^k)B_-(L^k)$$

- If we now include the possibility of failure, the update rule becomes:

$$B(L^k) = P(F^k = 0) * \eta P(o^k|L^k)B_-(L^k) + P(F^k = 1) * P(L^k|o^k)$$

- In practical terms this is done by replacing a number of samples in $B(L^k)$ equal to $P(F^k = 1) * n$ (where n = numer of samples) with samples from $P(L^k|o^k)$.
- The samples are replaced after the renormalization step to ensure that the first part of the equation is properly normalized by η.

Failure Recovery in Pseudocode

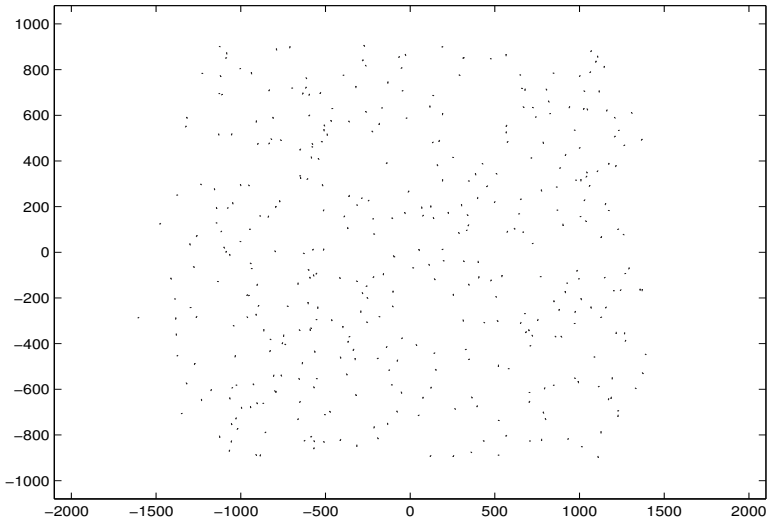
In pseudocode, the new sensor update rule looks like:

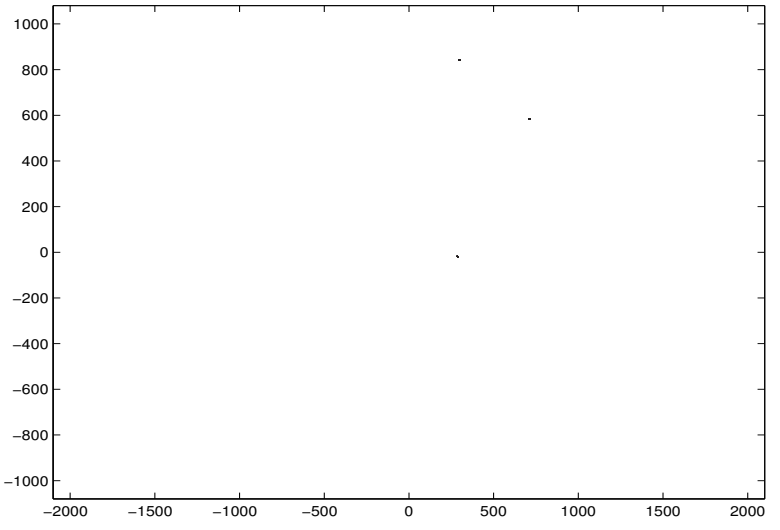
$p_s = 0$
For each sample $x^k_{-,i}$ in $B_-(L^k)$
    $w^k_i = P(o^k|L^k = x^k_{-,i}) * w^k_{-,i}$
    $p_s = p_s + w^k_i$
    Add $x^k_{-,i}$ with weight w^k_i to $B(L^k)$
If $p_s < p_t$
    $r = n * (1 - p_s/p_t)$
    Generate r samples y^k_j from $P(L^k|o^k)$
    Replace first r samples in $B(L^k)$ with new samples y^k_j.

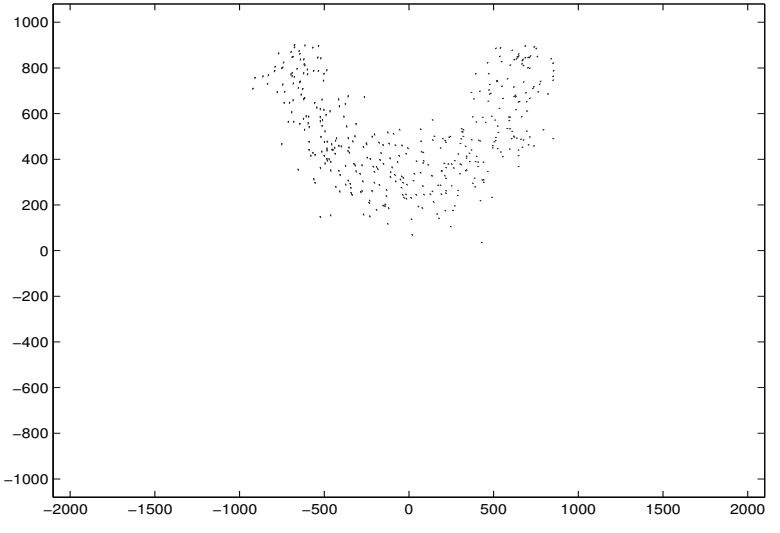
Failure Recovery Commentary

- Notice that in the normal case when things are working well, this acts exactly the same as the MCL update rule you saw earlier.
- This technique is called Sensor Resetting Localization (SRL) because it occasionally resets the localization based off of the sensor readings.
- This technique was introduced by Scott Lenser.

Failure Recovery in Action







Failure Recovery Effects

By implementing failure recovery, we get:

- Ability to use fewer samples since global localization no longer provides a bottle neck.
- Robustness to errors in the motion model.
- Ability to handle kidnapped robot problem.
- Robustness to unusual events.
- Quick recovery from unforeseen problems.

Extensions of SRL

- The work done in SRL has been extended by others in an algorithm known as Adaptive-MCL (A-MCL).
- The goal of this modification is to avoid the need to set a threshold probability.
- SRL can also be sensitive to particularly bad sensor readings.
- A-MCL is much less sensitive to this.
- The basic idea is to replace the calculation p_s/p_t with one based off actual measurements of p_s from when the localization is preforming properly.

A-MCL

- The calculation p_s/p_t which is used to determine the number of samples replaced is replaced in A-MCL.
- Instead, the calcuation $\nu * p_{\text{fast}}/p_{\text{slow}}$ is used.
- Here ν is a constant, typically 2.0.
- p_{slow} and p_{fast} are slow and fast exponential averages of p_s with p_{slow} being much slower than p_{fast}.
- A-MCL is currently used by our software on the AIBOs.
- $p_{\text{slow/fast}}$ is updated with $p_{\text{slow/fast}} = p_{\text{slow/fast}} * (1 - \gamma) + \gamma * p_s$
- We use $\gamma = .2$ for p_{fast} and $\gamma = .002$ for p_{slow}.

A-MCL Comments

- It takes several bad sensor readings to convince A-MCL that the localization has failed.
- This makes it robust to short sensor failures.
- Its adaptive nature makes it robust to shifts in the general level of likelihood of the sensors.
- Because samples are generated according to 2 exponential averages, it takes a while to switch from not generating samples to generating samples and back again.
- In practice, once a failure is detected. it tends to generate a small fraction of samples each sensor update until it has several sensor updates of evidence that the samples have improved the result.
- This results in effectively generating samples until they have an effect which results in very robust behavior.

SRL vs. A-MCL

- The newer A-MCL algorithm has some different behavior from SRL.
- It is much more robust to errors in the sensor model. Errors in the sensor model tend to make SRL reset too frequently.
- It is less committal on recovery, which prevents errors but can take longer to recover.
- SRL tends to generate samples in big bangs of a single update. A-MCL tends to generate samples in brief bursts over several updates.
- This is the main reason we are currently using A-MCL instead of SRL. It allows us to spread out the expensive operation of generating samples according to $P(L^t|o^t)$ across multiple frames.
- It is somewhat more robust than SRL.

Localization Code

- The interface is located in agent/Localization/LocalizationInterface.h
- The implementation of A-MCL is located in agent/Localization/SRL
- Environment.cc contains the map.
- Localization.cc contains interface routines.
- LocalizationEngine.cc contains the control logic.
- Primitives.cc contains the code to update samples for movement and weight them for sensors.
- Sampler.cc contains code to generate samples from the sensor readings.
- LocalizationEngine.h contains the constant numSamples which controls the number of samples used.

Localization Code

- The data structures related to the storage of the pose probability density samples are located in LocalizationEngine.h
- The implementation refers to a pose as a "locale".
- LocaleSampled stores the pose probability density sample points.
- It mostly contains an array of Samples of size numSamples.
- Sample represents one of the particles/samples used by the localization.
- It consists of a weight and a pose.
- The pose is represented as a 2d vector loc and an angle in radians.
- loc is relative to the origin of the map and angle is relative to the direction of the positive x-axis of the map.