Intelligent Robotics — Assignment Two

Jack Browne, Jack Jacques, Hugh Samson, and Matthew Williamson

4th November 2014

1 Experimentation

Our initial experimentation was to figure out the noise in the readings in order to adjust these values within the program and give the robot a better chance of knowing where it is. To do this we did separate experiments for the rotation, drift, and translation noise which are outlined below.

1.1 Translation Noise

For the translation noise we told the robot to move forward 0.5m/s for a defined amount of time. Using $speed \times time$ will then give you distance. e.g. $1m \equiv 0.5m/s \times 2s$. We then measured the actual result, and took the average of five tests at each distance. As is obvious there is around 2cm - 4cm of noise per meter.

Expected Result	Actual Result	Difference	Difference per Meter
1m	1.02m	0.02m	0.02m
2.5m	2.59m	0.09m	0.04m
5m	5.18m	0.18m	0.04m
7.5m	7.65m	0.15m	0.02m

1.2 Drift Noise

Drift noise was measured using the laser. We took a reading of how close to the wall it was, before moving it the designated distance and taking another reading. The absolute values of these readings were then averaged. However due to the difficult nature of aligning the robot *perfectly* it is possible that the experiment is flawed, especially at such negligible values.

Distance Travelled	Drift	Drift per Meter
1m	0.018m	0.018m
2m	0.035m	0.018m
3m	0.025m	0.008m

1.3 Rotation Noise

To get the actual value to turn we again told it to turn 90°a second, and stopped the rotation after the (supposed) correct rotation. As was obvious from our experiments there's very little noise over 90°. This is probably because the motors don't have enough time to engage, and then stop relatively quickly, and this leads to overturning by approximately 5°.

Expected Result	Actual Result	Diff	Diff in Radians	Noise per 360°
90°	95°	5°	0.087r	0.348r
180°	181°	1°	0.017r	0.034r
270°	271°	1°	0.017r	0.022r
360°	360°	0°	0r	0r

1.4 Estimate Pose Experimentation

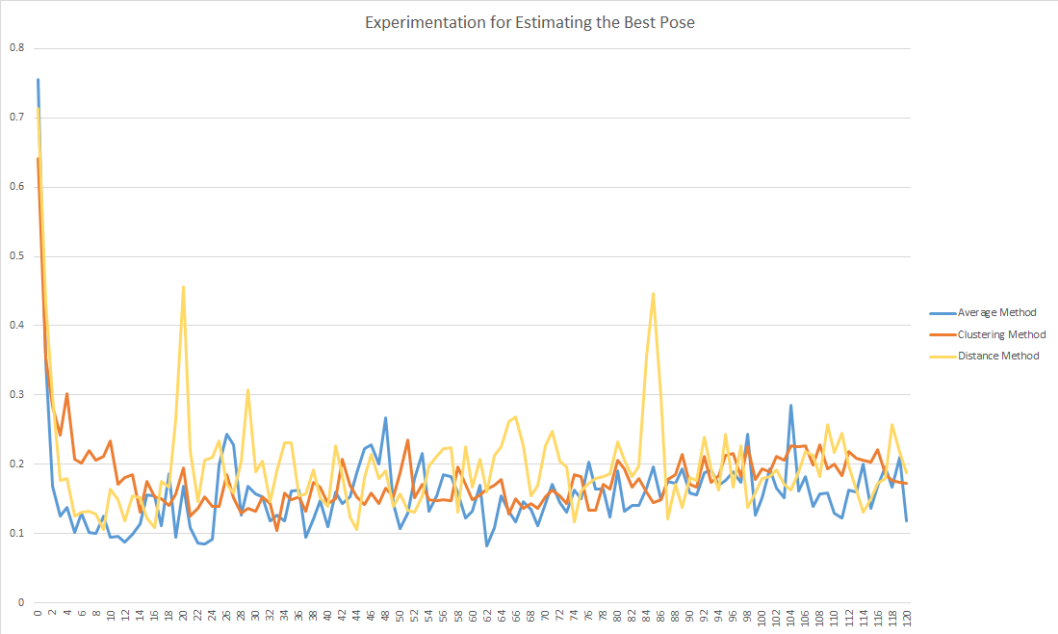
We also needed to experiment with the best method of estimating the pose, to do this we coded three methods to compare.

Distance Method A method which takes the sum of the distance between every other point - using math.hypot() - and returning the point with the minimum score

Clustering Method A method which calculates how many other points are within 0.5m of the point and returns the point with the minimum score

Average Method A method which takes the average of all points and returns a this new point

These methods were then run five times for two minutes, using our exercise one code to move the robot and the estimated pose was output along with the robot's actual position on Stage to a csv file. These tests were then averaged and the graph below produced.

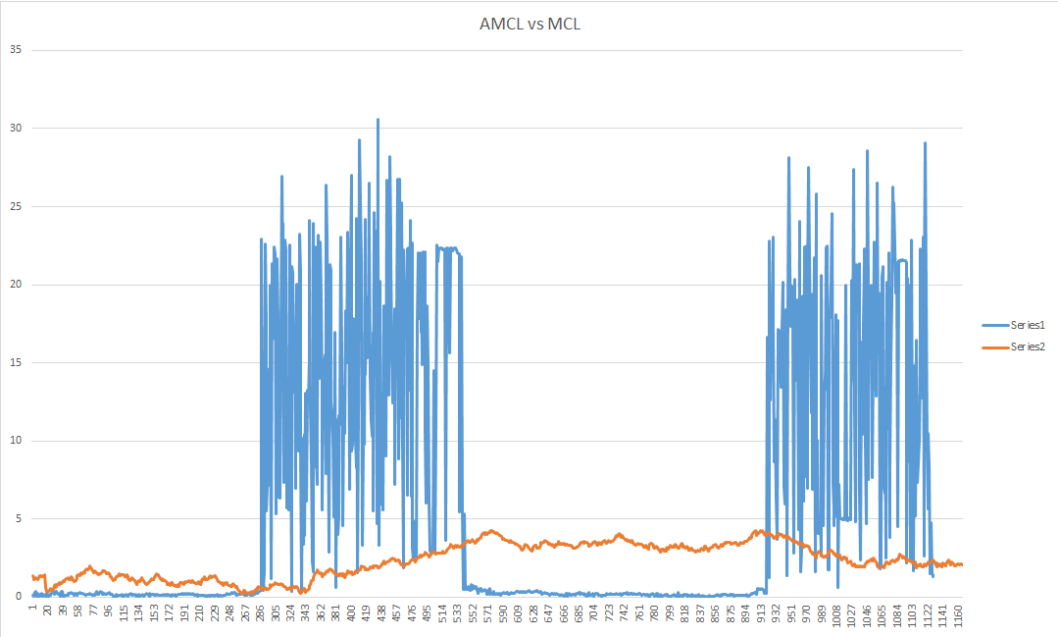


As you can see the **Distance Method** appears to be the least accurate. This could be because any outliers would heavily bias the 'best' point toward the outlier. Although the **Average Method** appears to be the most accurate it also varies a lot more than the **Clustering Method**, and because of this when run outside the simulator - and outside of a perfect action model - it may be that it performs worse. The **Clustering Method** appears to be the most stable, with few peaks or troughs after the initial localization, and therefore was the best in the test.

1.5 AMCL

Up to now all methods had been using standard Monte Carlo Localisation, however using tests shown below we decided that AMCL is a much neater solution, this is because it doesn't suffer from Particle Deprivation, and therefore can more easily correct itself when the particle cloud ends up in the wrong position.

vs mcl.png



Although the AMCL's estimated pose varies hugely while trying to relocalise it's obvious that after it has it's *much* more accurate.

2 Failed Solutions

2.1 Kullback-Leibler Divergence

We attempted to use KLD to improve the optimisation of our algorithm, so that we adapted computationally when our robot become unsure of its position. However, we didn't leave enough time to implement, and without a full implementation it actually hampered our `update_particle_cloud` method.

2.2 Adapting Computationally

We attempted to increase and decrease the size of our particle cloud based on how sure the robot was of being in a certain position, however this meant that it was more computationally intensive, and the `update_particle_cloud` method overran the timeslot by a significant amount - which led to problems with localisation.

3 Solution

3.1 initialise_particle_cloud(self, initialpose)

In this method we create a `PoseArray()`, and then fill it with random positions based on the `initialpose`. To do this we used `random.gauss()` and gave it a standard deviation - 0.5m for the x and y axis, and 0.125 radians (approximately 7 degrees) for the orientation. During testing this is also the method we started the robot moving, and started logging the robot's movement data using subprocesses.

3.2 update_particle_cloud(self, scan)

We used `numpy.cumsum()` to get the cumulative sum of all weights (from `self.sensor_model.get_weight()`) and stored the array as `breakpoints`, and the maximum in a separate variable. Then we calculated a random number (using `random.uniform()`) between 0 and the maximum, which was plugged into `bisect.bisect(breakpoints,maximum)`. This got us a random number - weighted toward the ' 'best' poses - which we then used as the base for a new pose. This new pose had some slight noise added using `random.gauss()` as in `initialise_particle_cloud()` and was stored in a new `PoseArray()`. After this had been done 200 times - or the length of the original `particlecloud.poses` - we replaced the original `particlecloud` with the new `PoseArray()`.

We found there was a trade off between accuracy and robustness using our noise values. Lower noise means that the `estimate_pose` method is more accurate, however a higher noise means that it's easier to react to significant changes in the odometry.

3.3 estimate_pose(self)

As shown in section 1.4 we did a fair bit of experimentation to decide on a suitable method for `estimate_pose()`. We chose the **Clustering Method** which is implemented as so. For every pose in `particlecloud.poses` we compare it against every other pose, and see whether those poses are within 0.5m. If they are then the '`pose_score` is incremented by 1. This continues until all poses have been checked against the initial pose - the `pose_score` is then stored in a dictionary with the pose as the key. After all poses have been iterated over then the pose with the highest `pose_score` is returned.