# Intelligent Robotics Exercise 2: *'Where am I?'*


UNIVERSITY OF BIRMINGHAM

## Team Leonard

Matthew Flint - 1247903 - mxf203@bham.ac.uk
Laurence Stokes - 0942846 - lls046@cs.bham.ac.uk
Robert Minford - 1189213 - rxm113@cs.bham.ac.uk
Alexander Bor - 1241885 - ahb285@cs.bham.ac.uk

*This work was conducted as part of the requirements of the Module 06-13520 Intelligent Robotics of the Computer Science department at the University of Birmingham, UK, and is submitted in accordance with the regulations of the University's code of conduct*

# Abstract

Localisation is a difficult task for an autonomous Robot. There are several methods and technologies available to the end of achieving localisation including *(but not limited to),* GPS, odometry (dead reckoning) and laser and sonar. In each approach, however, improvements in accuracy come at the cost of increasingly expensive hardware and additional processing power. This paper presents a method for localisation (specifically position tracking), as submitted in conformity with the requirements of Exercise Two of the 06-13520 Intelligent Robotics Module of the Computer Science Department at the University of Birmingham, UK. Our solution uses data from a laser sensor to calculate an estimate of the robot's pose (position and orientation), converging on the most likely possible pose using a Roulette Wheel Selection algorithm, as part of the MCL Framework for ROS.

# Table of Contents

# Introduction

Localisation is a difficult task for an autonomous Robot. There are several methods and technologies available to the end of achieving localisation including, but not limited to, GPS, odometry (dead reckoning[1]) and laser and sonar. In each approach, however, improvements in accuracy come at the cost of increasingly expensive hardware and additional processing power.

Our solution to the localisation problem (specifically *positioning tracking*) builds upon a Monte Carlo localization approach (as described by Dieter Fox) [1], which is included as part of the ROS ACML Framework. The Monte Carlo approach is an algorithm to determine a Robot's position and orientation (pose) using a particle filter, wherein each particle represents a possible pose of the Robot as a 3-dimensional vector $\{x,y,\theta\}$. The rationale for using a particle filter to the end of solving the localization problem is clear. Unlike Kalman Filters[2], particle filters can approximate a wide variety of probability distributions, not just normal distributions. They are also computationally efficient, as they focus resources on regions in state space with high likelihood [2].

A Roulette Wheel Selection algorithm is applied to the particle filter to stochastically sample and repeatedly update the the particle cloud by probabilistically replacing poses with smaller weightings for those with higher weightings. After enough iterations, the actual pose of the Robot should be converged on with complete accuracy. Included in this report is our solution to the task as well as our testing and experimental results, which are presented and analysed in the discussion section.

# Solution

The first step to implementing a working solution was to get a good grasp of the supplied boilerplate code, as well as what was needed to get it working. Contained in the code were three method declarations with no method bodies; they were to be filled in as per the exercise. The first method, initliaze_particle_cloud, computes estimates of where the expected view of the robot approximately matches the observed range of the sensor data,

---

[1] The process of calculating one's current position by using a previously determined position, or fix, and advancing that position based upon known or estimated speeds over elapsed time and course; for example using the number of wheel revolutions of the robot. This is an idiothetic source and whilst it can give the absolute position of the robot, it is subject to cumulative error which can grow quickly.

[2] An algorithm that uses a series of measurements observed over time, containing noise (random variations) and other inaccuracies, and produces estimates of unknown variables that tend to be more precise than those based on a single measurement alone.

setting up an array of possible poses with likelihood weightings. The second method, update_particle_cloud, contains the logic for the roulette wheel selection, and the third method, estimate_pose, attempts to calculate the actual pose of the Robot.

## The Positioning Algorithm

The positioning algorithm we adopted was a Roulette Wheel Selection, aka *Fitness proportionate selection,* which is a genetic algorithm for selecting more useful (or 'fit') solutions for recombination [3]. In our solution, this meant that poses in the cloud with higher weights were more likely to be selected, with the intended result of the particle cloud ultimately converging on the Robot's actual pose. Our algorithm was as follows:

```
For pose in range 0 to x (where x is the number of poses in the particle cloud):
       Sum the weights of all the poses in the existing particle cloud
       Initialize a list containing the cumulative sum of (pose weight/sum of pose weights)

Generate a random floating point number between 0 and 1.
Initialise a new particle cloud

For pose in range 0 to x (where x is the number of poses in the particle cloud)
       select the pose in the position of the cumulative sum list that corresponds to the
       random number generated, and insert that into a new particle cloud.
```

The full python implementation of the algorithm can be found in the Appendices.

## Noise

Whilst the pose of the Robot is determined by the sensor data, that data is often imperfect and subject to 'noise'. Noise is random deviation of the sensor data that varies with time, and usually occurs because of inaccuracy in hardware sensing and transmission [4]. Noise that we accounted for in our data was split into three sections from the odometry motion model:

```
ODOM_ROTATION_NOISE =  ... #Odometry model rotation noise
ODOM_TRANSLATION_NOISE = ... #Odometry model x axis (forward) noise
ODOM_DRIFT_NOISE = ... #Odometry model y axis (side-to-side) noise
```

We accounted for the noise by tracking the predicted vs actual output of the Robot's movement. For example, if the odometer reports the Robot has moved 100cm (1 meter), but the the movement is only 96cm, we can assume 4cm (4%) of noise affecting the odometer translation in the respective axis.

# Discussion

## Issues

Our group suffered several obstacles to progression, specifically in the transition from localisation in the rviz simulation versus the ‘real-world’’. In real-world testing, the laser scanner was reporting ‘inf’  results, a bug which took significant time to overcome and fix. Nonetheless, we managed to overcome this (with help from Bruno Lacerda) by having a guard check for inf values and, if any were reported, setting them to be equal the maximum value of the current scan (rather than subsequently process them and encountering resulting errors).

## Testing

Once the positioning algorithm was implemented, we conducted several tests to ensure both the accuracy of the method and that sensor noise was successfully and sufficiently accounted for.

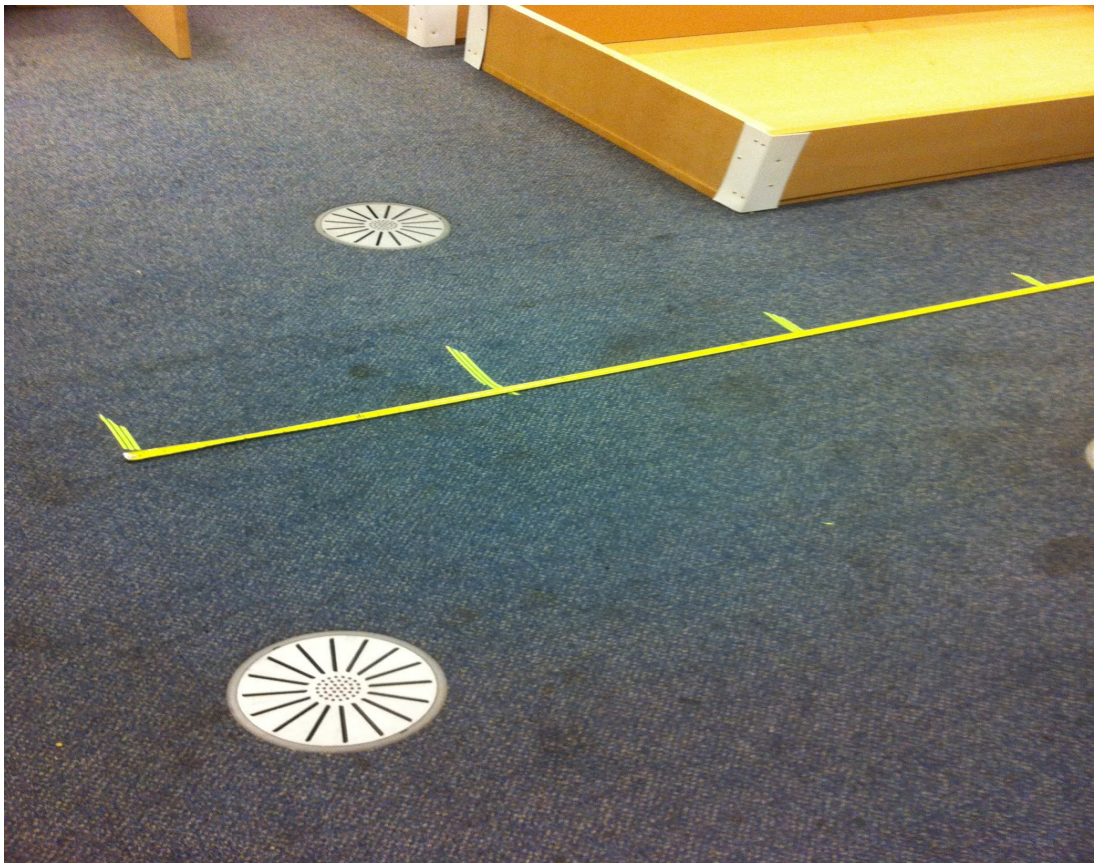
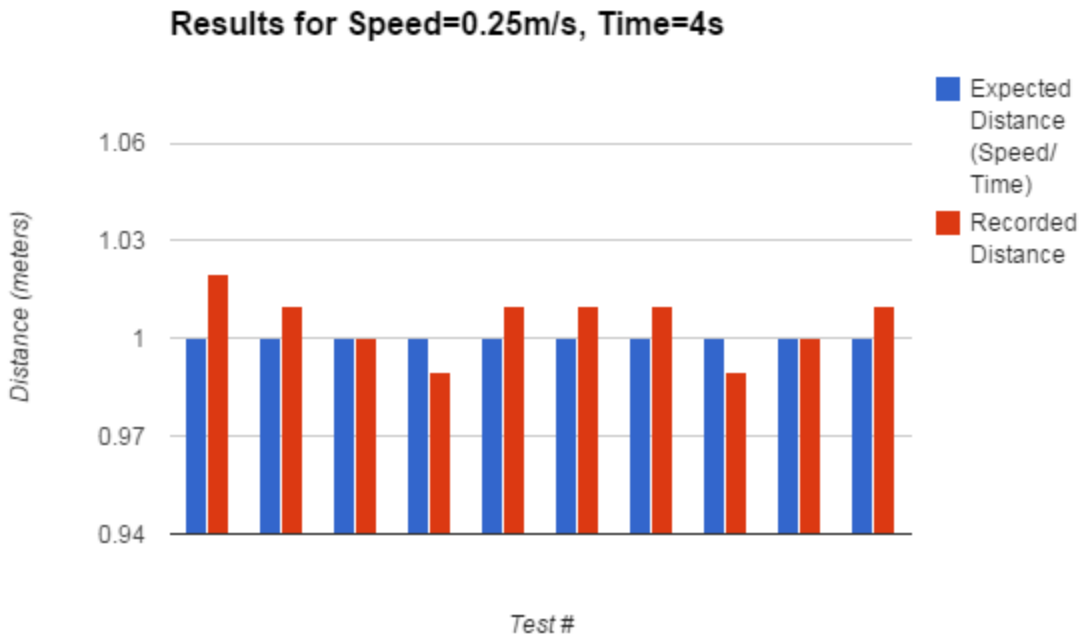


*Figure 1: Robot Lab floor with marked distances for testing odometer (translation) noise*

In regards to noise testing, we were primarily looking for the discrepancies in the reported position of the Robot from the position tracking algorithm versus the reported position from the odometer. To this end we conducted a series of tests and averaging the results of the tests (respective to the specific odometer noise axis) gave an indication of the level of noise to expect (and thus to account for) from the odometer. It should be noted, however, that our testing was subject to human error. Eliminating this error vector would have demanded more complex testing methods beyond the scope of the exercise.

## Translation Noise

The first series of tests we conducted were to investigate the translation - *the y axis, or forwards* - noise. This involved driving the Robot forwards[3], with a set velocity and time, expecting the distance travelled to be speed x time. We then measured the actual distance travelled versus the expected distance. The results of the 30 test instances indicated that the odometer translation noise was on average ~0.7%, with this value fairly consistent across a variety of times when using a value of 0.25 for speed. Results are presented in graphs below and full details of the test results can be found in the appendices.




[3] It should be noted that the motor of the Robot has a short delay before starting. We tested this to consistently be ~2 seconds (+- 0.1) and accounted for it by initializing the motor (by sending a spurious command) prior to sending a movement command.


Results for Speed=0.25m/s, Time=8s
Expected Distance (Speed/ Time)
Recorded Distance
2.04
2.02
2
1.98
1.96
Distance (meters)
Test #

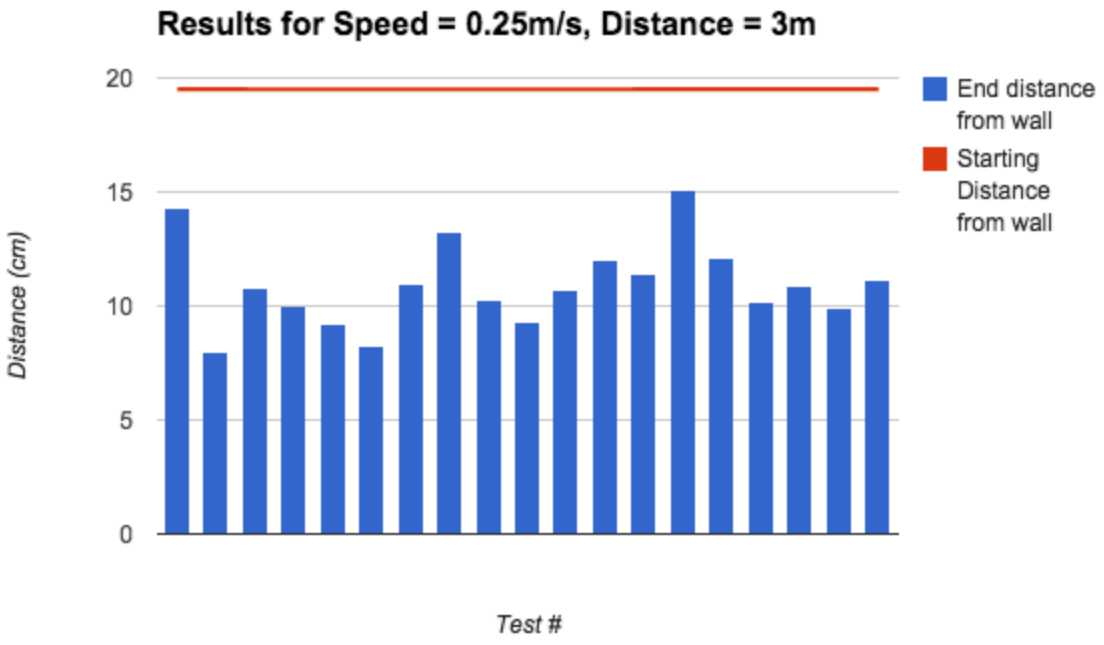

Results for Speed=0.25m/s, Time=12s
Expected Distance (Speed/ Time)
Recorded Distance
3.06
3.03
3
2.97
2.94
Distance (meters)
Test #

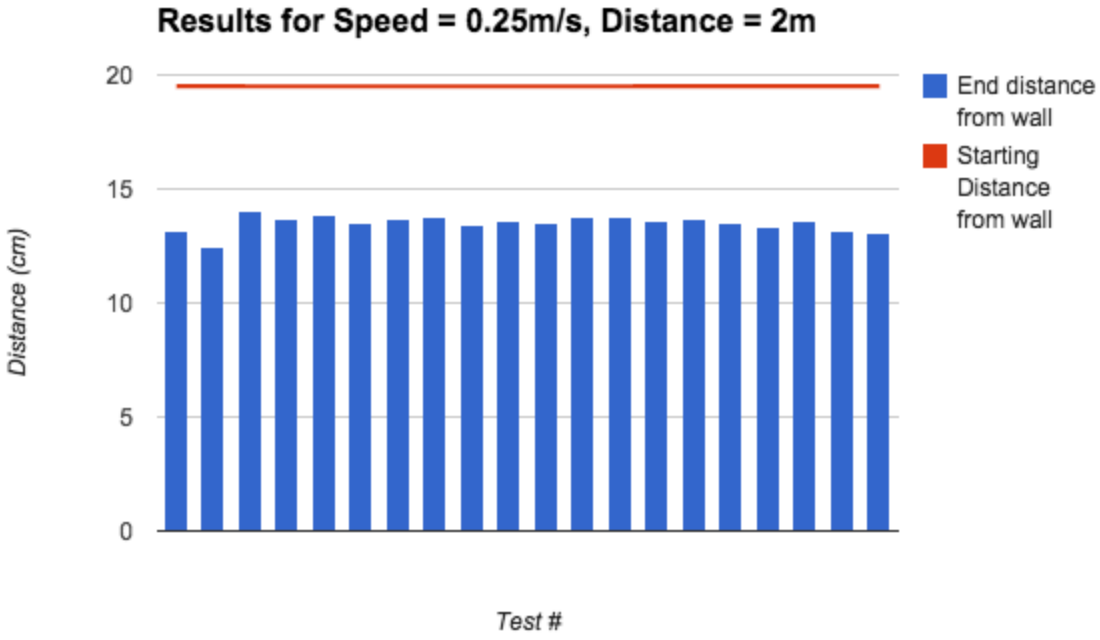
### Drift Noise

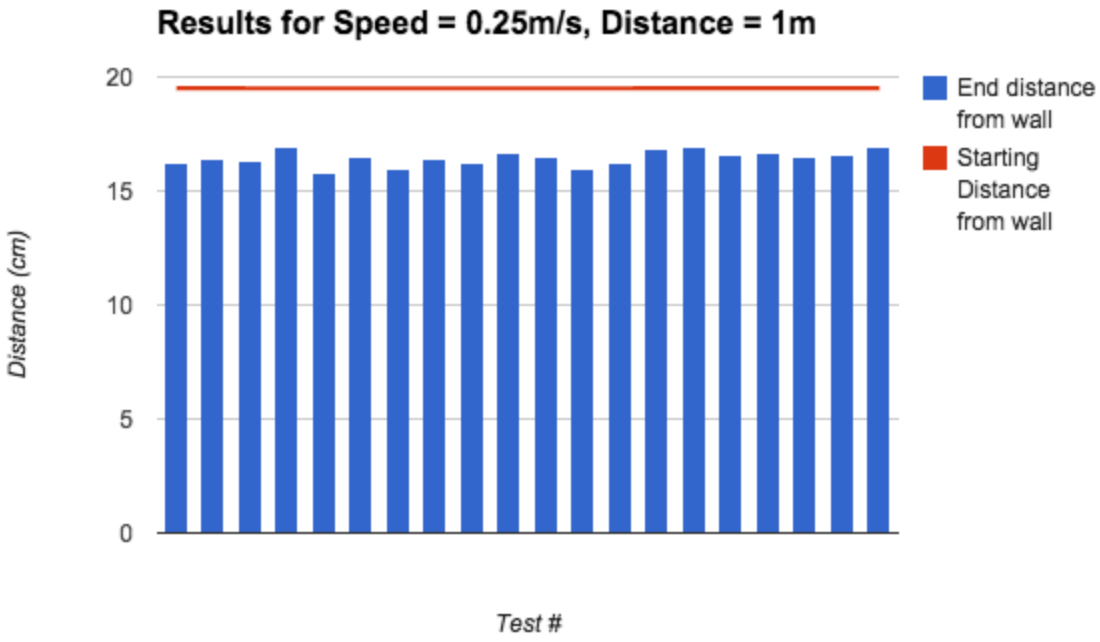
The second series of tests we conducted were for the odometer drift noise, and conducted similarly to the tests for the translation noise. Specifically, we instructed the Robot to drive 1, 2, or 3 meters on a predefined straight path, and then measured the x-axis deviation from the path; repeating this 20 times for each distance.

Our results showed an average drift per meter of 2.9159 cm/m for the 60 tests at the 3 different distances (2.7125, 2.99 and 3.045 for 3 meters, 2 meters and 1 meter respectively). This means that for each meter the robot drives forwards, it will, on average, drift ~2.9159cm to the left. The results also suggest that the drift noise decreases as distance increases. Further testing would be required to test this hypothesis, however, as our results could be anomalous, or more likely specific to the small number of distances we tested (i.e. the phenomena is only observable over small distances).

Detailed results are presented in graphs below and complete details of the test results can be found in the appendices.



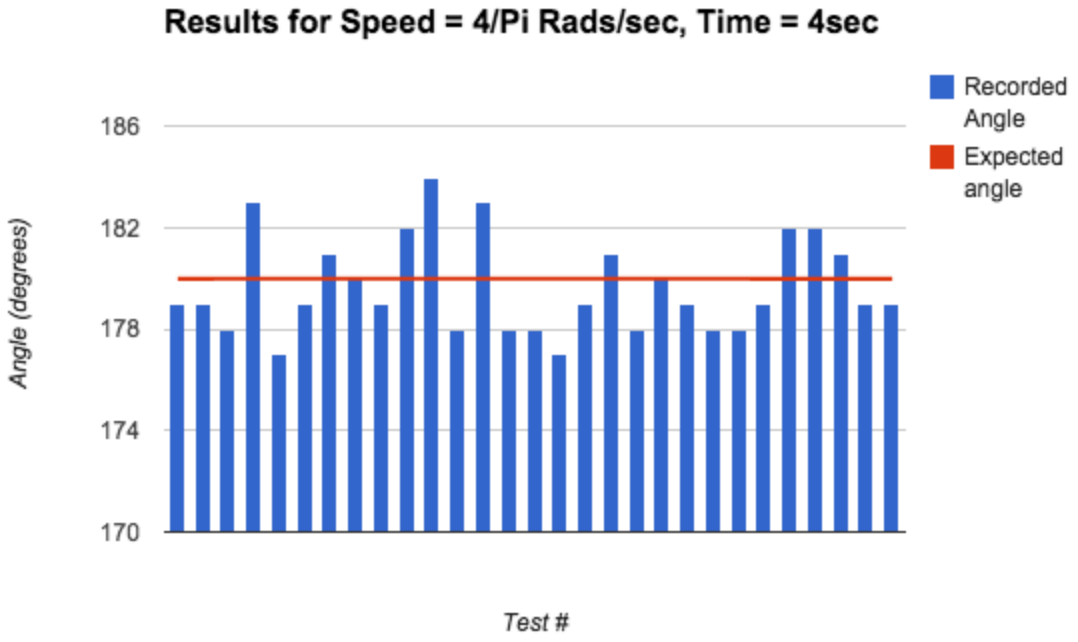

Results for Speed = 0.25m/s, Distance = 2m
20
15
10
5
0
Distance (cm)
Test #
End distance from wall
Starting Distance from wall


Results for Speed = 0.25m/s, Distance = 1m
20
15
10
5
0
Distance (cm)
Test #
End distance from wall
Starting Distance from wall

### Rotation Noise

The final series of tests we conducted were for the odometer rotation noise. To test rotation noise we set the Robot to turn a set distance ($\pi$ radians - 180 degrees) at a set angular.z velocity (turning speed) of $4/\pi$ radians/second (meaning it was turning at 45 degrees/second). We then measured the actual degrees turned over the 4 seconds and compared the difference. We found, over 30 test instances, the average difference to be ~0.55%, with a variance of +-4 degrees.

As with the analysis of translation and drift noise test results, detailed results for the rotation noise tests are presented in graphs below and complete details of the test results can be found in the appendices




## Improvements

The obstacles to development notwithstanding,  we recognise we could have improved our methods and thus our localisation strategy. In the estimate_pose method, for example, we simply calculated the average of all the poses, as opposed to firstly discounting outlying values before making any calculations.

# Conclusion

We have presented a Roulette Wheel Selection position tracking approach for the pioneer robot that maintains reliability even in the face of noisy sensor information. We have overcome and discussed issues to progress and have evaluated our results vis-a-vis issues, testing, and improvements, and can demonstrably prove the success of the implementation.

# Acknowledgements

The authors of the paper would like to thank the module convenors and demonstrators for their help and guidance throughout this exercise. Specifically, we would like to thank Bruno Lacerda, for his thorough help in overcoming the inf bug we were challenged with. Without his help this report would have not been completed to the standard that it is.

# References

1. *'AMCL Package Summary',* ROS.org, Available: http://wiki.ros.org/amcl [Online] (Accessed 21/10/2014)

2. *'Particle Filters for Robot Localisation'*, D. Fox et al. Available: lhttp://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=547840CC2F7B99B070728AE1B1531BA1?doi=10.1.1.1.9914&rep=rep1&type=pdf [Online] (Accessed 27/10/2014)

3. *'Roulette Wheel Selection'*, Newcastle University Engineering Design Center, Available: http://www.edc.ncl.ac.uk/highlight/rhjanuary2007g02.php/ [Online] (Accessed 27/10/2014)

4. *'SensoClean: Handling Noisy and Incomplete Data in Sensor Networks using Modelling*', http://www.sccs.swarthmore.edu, Available:http://www.sccs.swarthmore.edu/users/03/yeelin/docs/finalreport.pdf [Online] (Accessed 27/10/2014)

# Appendices

## Appendix 1 Python Code

```
#-------------- Imports -------------#

from geometry_msgs.msg import Pose, PoseArray, Quaternion
from pf_base import PFLocaliserBase
import math
import numpy.ma as ma
import rospy
from util import rotateQuaternion, getHeading
import random
from time import time


#-------------- PFLocaliser Class Methods-------------#

class PFLocaliser(PFLocaliserBase):

    #constructor
    def __init__(self):

        #Call the superclass constructor
        super(PFLocaliser, self).__init__()

        #Set motion model parameters
        self.ODOM_ROTATION_NOISE = 0.005
        self.ODOM_TRANSLATION_NOISE = 0.014
        self.ODOM_DRIFT_NOISE = 0.006

        #Sensor model readings
        self.NUMBER_PREDICTED_READINGS = 100


    def initialise_particle_cloud(self, initialpose):
        #Set particle cloud to initialpose plus noise

        noise = 1
        #create an array of Poses
        posArray = PoseArray()

        #iterate over the number of particles and append to PosArray
        for x in range(0, self.NUMBER_PREDICTED_READINGS):
            pose = Pose()

            pose.position.x =  random.gauss(initialpose.pose.pose.position.x, noise)
```

```
            pose.position.y = random.gauss(initialpose.pose.pose.position.y, noise)
            pose.position.z = 0

            pose.orientation = rotateQuaternion(initialpose.pose.pose.orientation,
math.radians(random.gauss(0,15)))

            posArray.poses.extend([pose])

        #print posArray
        return posArray

    def update_particle_cloud(self, scan):
        '''method called whenever a new LaserScan message
           is recieved. Does the particle filtering'''

        print("Running update_particle_cloud")

        #list of poses
        sum_weights = 0
        cumulative_sum_list = []
        segment_list = []
        sum_count = 0
        y=0

        pcloud = self.particlecloud.poses

        #print scan
        print len(scan.ranges)
        scan.ranges=ma.masked_invalid(scan.ranges).filled(scan.range_max)
        print len(scan.ranges)

        #Calculate the sum of the weights
        for particle in pcloud:
            particleWeight = self.sensor_model.get_weight(scan, particle)
            sum_weights+= particleWeight

        #Create a list of the cumulative sum of the weights
        for particle in pcloud:
            particleWeight = self.sensor_model.get_weight(scan, particle)
            weight_over_sum = particleWeight/sum_weights
            segment_list.extend([weight_over_sum])
            sum_count+= weight_over_sum
            cumulative_sum_list.extend([sum_count])

        #new pose array to replace partical cloud
        newParticleCloud = PoseArray()

        #Select particles with higher weight to be copied into
        #the new particle cloud
        for particle in pcloud:
```

```
            rand = random.uniform(0,1)
            segment_count = 0
            found = False
            output = 0
            for x in cumulative_sum_list:
                #if x is greater than rand
                if rand <= x:
                    output = segment_count
                    found = True
                if found:
                    #print pcloud[output]
                    break
                segment_count+=1
            newParticleCloud.poses.extend([pcloud[output]])

        '''the weighting of the particle at the segment_list we
        are in. Higher weighted particles have bigger segments
        and so are more likely to be output'''

        newPcloud = PoseArray()

        #Add noise
        for particle in newParticleCloud.poses:
            newPose = Pose()
            newPose.position.x = random.gauss(particle.position.x,(particle.position.x *
self.ODOM_DRIFT_NOISE))
            newPose.position.y = random.gauss(particle.position.y,(particle.position.y *
self.ODOM_TRANSLATION_NOISE))
            newPose.orientation = rotateQuaternion(particle.orientation,
math.radians(random.uniform(-self.ODOM_ROTATION_NOISE,self.ODOM_ROTATION_NOISE)))

            newPcloud.poses.extend([newPose])

        #Set self.particlecloud to new particlecloud
        self.particlecloud = newPcloud




    def estimate_pose(self):
        'Method to estimate the pose'
        '''Create new estimated pose, given particle cloud
        E.g. just average the location and orientation values of each of
        the particles and return this.

        Better approximations could be made by doing some simple clustering,
        e.g. taking the average location of half the particles after
        throwing away any which are outlier'''
```

```
#declare some variables
x,y,z,orix,oriy,oriz,oriw,count = 0,0,0,0,0,0,0,0

#iterate over each particle extracting the relevant
#averages
for particle in self.particlecloud.poses:
    x += particle.position.x
    y += particle.position.y
    z += particle.position.z
    orix += particle.orientation.x
    oriy += particle.orientation.y
    oriz += particle.orientation.z
    oriw += particle.orientation.w

count = len(self.particlecloud.poses)

#create a new pose with the averages of the location and
#orientation values of the particles
pose = Pose()

pose.position.x = x/count
pose.position.y = y/count
pose.position.z = z/count

pose.orientation.x = orix/count
pose.orientation.y = oriy/count
pose.orientation.z = oriz/count
pose.orientation.w = oriw/count

return pose
```

## Appendix 2 Translation Noise Testing Full Results

| Experiment # | | Speed | Time | Expected Distance (Speed/Time) | Recorded Distance | Percentage Difference |
|---|---|---|---|---|---|---|
| | | | | | | |
| 1 | | 0.25 | 4 | 1 | 1.02 | 2 |
| 2 | | 0.25 | 4 | 1 | 1.01 | 1 |
| 3 | | 0.25 | 4 | 1 | 1 | 0 |
| 4 | | 0.25 | 4 | 1 | 0.99 | 1 |
| 5 | | 0.25 | 4 | 1 | 1.01 | 1 |
| 6 | | 0.25 | 4 | 1 | 1.01 | 1 |
| 7 | | 0.25 | 4 | 1 | 1.01 | 1 |
| 8 | | 0.25 | 4 | 1 | 0.99 | 1 |
| 9 | | 0.25 | 4 | 1 | 1 | 0 |
| 10 | | 0.25 | 4 | 1 | 1.01 | 1 |
| 11 | | 0.25 | 8 | 2 | 1.99 | 0.5 |
| 12 | | 0.25 | 8 | 2 | 2.01 | 0.5 |
| 13 | | 0.25 | 8 | 2 | 2.02 | 1 |
| 14 | | 0.25 | 8 | 2 | 2.01 | 0.5 |
| 15 | | 0.25 | 8 | 2 | 2.02 | 1 |
| 16 | | 0.25 | 8 | 2 | 2 | 0 |
| 17 | | 0.25 | 8 | 2 | 2.02 | 1 |
| 18 | | 0.25 | 8 | 2 | 2.01 | 0.5 |
| 19 | | 0.25 | 8 | 2 | 2.01 | 0.5 |
| 20 | | 0.25 | 8 | 2 | 1.99 | 0.5 |
| 21 | | 0.25 | 12 | 3 | 3.03 | 1 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 22 | | 0.25 | 12 | 3 | 3.03 | 1 |
| 23 | | 0.25 | 12 | 3 | 3.01 | 0.3333333333 |
| 24 | | 0.25 | 12 | 3 | 3.02 | 0.6666666667 |
| 25 | | 0.25 | 12 | 3 | 3 | 0 |
| 26 | | 0.25 | 12 | 3 | 3.03 | 1 |
| 27 | | 0.25 | 12 | 3 | 3.01 | 0.3333333333 |
| 28 | | 0.25 | 12 | 3 | 2.99 | 0.3333333333 |
| 29 | | 0.25 | 12 | 3 | 3.02 | 0.6666666667 |
| 30 | | 0.25 | 12 | 3 | 3.03 | 1 |

# Appendix 3 Drift Noise Testing Full Results

| Distance | Starting Distance from wall | End distance from wall | Distance Difference |
|---|---|---|---|
| 3 | 19.5 | 14.3 | 5.2 |
| 3 | 19.5 | 8 | 11.5 |
| 3 | 19.5 | 10.75 | 8.75 |
| 3 | 19.5 | 10 | 9.5 |
| 3 | 19.5 | 9.25 | 10.25 |
| 3 | 19.5 | 8.25 | 11.25 |
| 3 | 19.5 | 11 | 8.5 |
| 3 | 19.5 | 13.25 | 6.25 |
| 3 | 19.5 | 10.25 | 9.25 |
| 3 | 19.5 | 9.3 | 10.2 |
| 3 | 19.5 | 10.7 | 8.8 |
| 3 | 19.5 | 12.05 | 7.45 |
| 3 | 19.5 | 11.4 | 8.1 |
| 3 | 19.5 | 15.1 | 4.4 |
| 3 | 19.5 | 12.1 | 7.4 |
| 3 | 19.5 | 10.15 | 9.35 |
| 3 | 19.5 | 10.9 | 8.6 |
| 3 | 19.5 | 9.9 | 9.6 |
| 3 | 19.5 | 11.1 | 8.4 |
| | | | |
| 2 | 19.5 | 13.2 | 6.3 |
| 2 | 19.5 | 12.5 | 7 |
| 2 | 19.5 | 14 | 5.5 |
| 2 | 19.5 | 13.7 | 5.8 |
| 2 | 19.5 | 13.9 | 5.6 |
| 2 | 19.5 | 13.5 | 6 |
| 2 | 19.5 | 13.7 | 5.8 |
| 2 | 19.5 | 13.8 | 5.7 |
| 2 | 19.5 | 13.4 | 6.1 |
| 2 | 19.5 | 13.6 | 5.9 |

| | | | |
|---|---|---|---|
| 2 | 19.5 | 13.5 | 6 |
| 2 | 19.5 | 13.8 | 5.7 |
| 2 | 19.5 | 13.8 | 5.7 |
| 2 | 19.5 | 13.6 | 5.9 |
| 2 | 19.5 | 13.7 | 5.8 |
| 2 | 19.5 | 13.5 | 6 |
| 2 | 19.5 | 13.3 | 6.2 |
| 2 | 19.5 | 13.6 | 5.9 |
| 2 | 19.5 | 13.2 | 6.3 |
| 2 | 19.5 | 13.1 | 6.4 |
| | | | |
| 1 | 19.5 | 16.2 | 3.3 |
| 1 | 19.5 | 16.4 | 3.1 |
| 1 | 19.5 | 16.3 | 3.2 |
| 1 | 19.5 | 16.9 | 2.6 |
| 1 | 19.5 | 15.8 | 3.7 |
| 1 | 19.5 | 16.5 | 3 |
| 1 | 19.5 | 16 | 3.5 |
| 1 | 19.5 | 16.4 | 3.1 |
| 1 | 19.5 | 16.2 | 3.3 |
| 1 | 19.5 | 16.7 | 2.8 |
| 1 | 19.5 | 16.5 | 3 |
| 1 | 19.5 | 16 | 3.5 |
| 1 | 19.5 | 16.2 | 3.3 |
| 1 | 19.5 | 16.8 | 2.7 |
| 1 | 19.5 | 16.9 | 2.6 |
| 1 | 19.5 | 16.6 | 2.9 |
| 1 | 19.5 | 16.7 | 2.8 |
| 1 | 19.5 | 16.5 | 3 |
| 1 | 19.5 | 16.6 | 2.9 |
| 1 | 19.5 | 16.9 | 2.6 |

# Appendix 4 Rotation Noise Testing Full Results

| Rotation Speed | Rotation Time | Expected angle | Recorded Angle | Angle Difference | % Difference |
|---|---|---|---|---|---|
| 4/Pi | 4 | 180 | 179 | -1 | -0.5555555556 |
| 4/Pi | 4 | 180 | 179 | -1 | -0.5555555556 |
| 4/Pi | 4 | 180 | 178 | -2 | -1.111111111 |
| 4/Pi | 4 | 180 | 183 | 3 | 1.666666667 |
| 4/Pi | 4 | 180 | 177 | -3 | -1.666666667 |
| 4/Pi | 4 | 180 | 179 | -1 | -0.5555555556 |
| 4/Pi | 4 | 180 | 181 | 1 | 0.5555555556 |
| 4/Pi | 4 | 180 | 180 | 0 | 0 |
| 4/Pi | 4 | 180 | 179 | -1 | -0.5555555556 |
| 4/Pi | 4 | 180 | 182 | 2 | 1.111111111 |
| 4/Pi | 4 | 180 | 184 | 4 | 2.222222222 |
| 4/Pi | 4 | 180 | 178 | -2 | -1.111111111 |
| 4/Pi | 4 | 180 | 183 | 3 | 1.666666667 |
| 4/Pi | 4 | 180 | 178 | -2 | -1.111111111 |
| 4/Pi | 4 | 180 | 178 | -2 | -1.111111111 |
| 4/Pi | 4 | 180 | 177 | -3 | -1.666666667 |
| 4/Pi | 4 | 180 | 179 | -1 | -0.5555555556 |
| 4/Pi | 4 | 180 | 181 | 1 | 0.5555555556 |
| 4/Pi | 4 | 180 | 178 | -2 | -1.111111111 |
| 4/Pi | 4 | 180 | 180 | 0 | 0 |
| 4/Pi | 4 | 180 | 179 | -1 | -0.5555555556 |
| 4/Pi | 4 | 180 | 178 | -2 | -1.111111111 |
| 4/Pi | 4 | 180 | 178 | -2 | -1.111111111 |
| 4/Pi | 4 | 180 | 179 | -1 | -0.5555555556 |
| 4/Pi | 4 | 180 | 182 | 2 | 1.111111111 |
| 4/Pi | 4 | 180 | 182 | 2 | 1.111111111 |
| 4/Pi | 4 | 180 | 181 | 1 | 0.5555555556 |
| 4/Pi | 4 | 180 | 179 | -1 | -0.5555555556 |
| 4/Pi | 4 | 180 | 179 | -1 | -0.5555555556 |