Mobile Robots Assignment 1

Charlie Maclean

January 2020

Exercise 1

(a)

$$\frac{dx}{dt} = u \cdot cos(\theta)$$

$$\frac{dy}{dt} = u \cdot sin(\theta)$$

$$\frac{d\theta}{dt} = \omega$$

(b) Euler's method says given $\frac{dx}{dt} = f(t)$ and $x(t_0) = x_0$, you can approximate $x(t_0 + h) \approx x_0 + h \cdot f(h)$. In the context of this question, we produce the following equations:

$$x(t + dt) = x(t) + dt \cdot (u \cdot cos(\theta))$$

$$y(t + dt) = y(t) + dt \cdot (u \cdot sin(\theta))$$

$$\theta(t + dt) = \theta(t) + dt \cdot \omega$$

(c) Trajectories are in Figure 1. We are simulating the motion of a robot using Euler integration to predict the change in motion between steps. We plot at various different step sizes.

In general, the smaller the step size, the more accurate the simulation is, as Euler's method has an error each step of approximately $O(dt^2)$. Hence, the lower the step size, the lower the error.

However, the smaller the step size the more calculations we must make, and hence the more computationally difficult the task becomes.

(d) Consider $z_n = (x_n, y_n, \theta_n)$ to be the state vector at step n. We then approximate the next state vector z_{n+1} using the current state vector, t_n - the time at step n, and dt - the time between steps.

We can find the derivative $\frac{dz}{dt}$, which is a function $f(t, z)$, as follows:

$$\frac{dz}{dt} = f(t, z) = (u \cdot cos(z_\theta), u \cdot sin(z_\theta), cos(t))$$

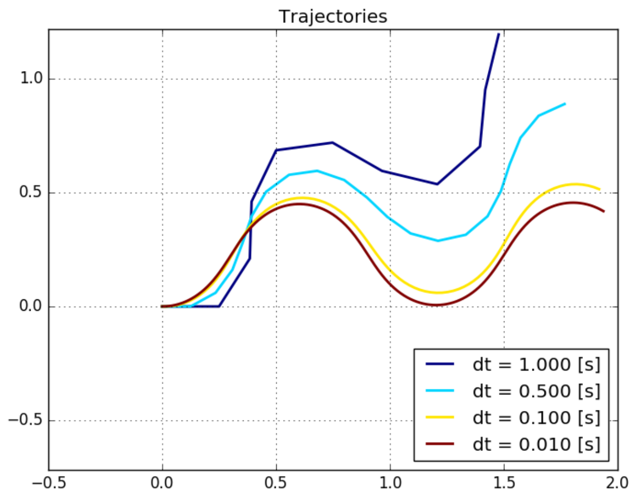


Figure 1: Trajectories with Euler Integration

Now, the RK4 approximation is used. Intuitively, it can be thought of as using a weighted average of gradients across the curve to provide a more accurate next step. The calculations are as follows:

$$k_1 = dt \cdot f(t_n, z_n)$$

$$k_2 = dt \cdot f(t_n + \frac{dt}{2}, z_n + \frac{k_1}{2})$$

$$k_3 = dt \cdot f(t_n + \frac{dt}{2}, z_n + \frac{k_2}{2})$$

$$k_4 = dt \cdot f(t_n + dt, z_n + k_3)$$

$$z_{n+1} = z_n + \frac{1}{6} \cdot (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4)$$

(e) Trajectories using the RK4 method are displayed in Figure 2.

Euler is a first order method, RK4 is fourth order. Euler's method estimates the next value based on one gradient, whereas RK4 uses a weighted average of four gradients. Hence, RK4 is more computationally expensive, as for each step it uses approximately four times the resources.

However, as we can see in the trajectories plotted, RK4 is much more accurate. Mathematically this can be explained as for small timesteps dt, the error per step of Euler is $O(dt^2)$, whereas the error per step of RK4 is $O(dt^5)$. This increase of accuracy means we see that RK4, unlike Euler, doesn't diverge from the expected trajectory, even with the largest

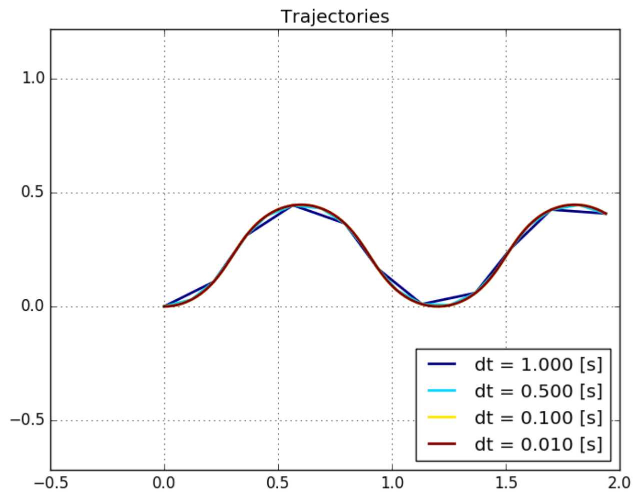


Figure 2: Trajectories with RK4 Integration

time-step of one second. Therefore, RK4 is useful even at large timesteps, unlike Euler.

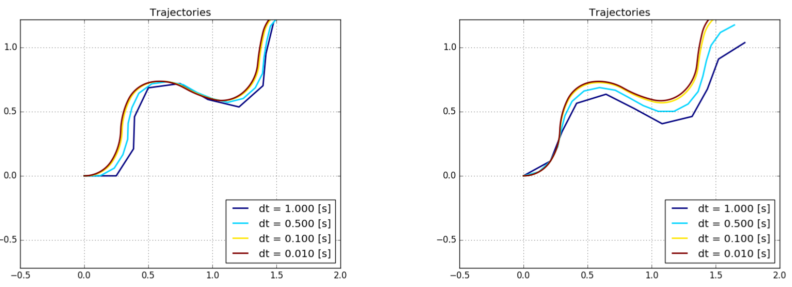
Euler may be practical in a few cases - if we have very simple hardware it will be easier to calcualate Euler. Additionally, if we are using a very small time-step anyway, the two methods are similar in terms of accuracy so opting for Euler makes sense.

(f) The plots of the trajectories with the new 1hz perception-action loop are in Figure 3.

Both methods produce a cosine graph which is pointed with a positive gradient. This is correct, and comes as a result of the new value of ω, which stays at $cos(0) = 1$ for an entire second. This high initial value skews the entire trajectory upwards.

Euler captures the trajectory pretty accurately at each time-step. This is a result of Euler method's assumption that the derivative won't change over the step - an assumption which is true for θ in this case.

On the other hand, RK4 has a tendency to under-estimate the curves, particularly at high time-step values. This is as RK4 makes use of a weighted average of the gradient at different points along the trajectory. Some of the gradients it computes in this average will cross the threshold of floor, meaning the weighted average is skewed from the actual gradient between the steps. In particular, the graphs are under-estimated because in the range $0 \leq t \leq 1$, RK4 is considering the gradient of θ above $t = 1$, which is lower than it should be.



(a) Euler (b) RK4

Figure 3: Trajectories with a 1hz Perception-Action Loop

(g) Fixed time steps are problematic in the case where we have a function with a derivative that varies a lot. Large time-steps are required when the derivative is constant to save resources, but we need small time-steps when the derivative changes a lot, as otherwise we lose a lot of accuracy. With a fixed time step, we are forced to choose between accuracy or ease of computation. However, there is a better way - with an adaptive time-step we can vary the time-step such that we get accuracy at highly changeable regions and efficiency otherwise.

In question f, we have a case of a discontinuous function - the gradient of θ is constant, until it reaches the next integer, when it changes instantly. Hence, we could do with an adaptive step size, where we have more steps around the region of discontinuity - around the integers.

My solution was to insert a new value before the region of discontinuity - before every integer, I introduce a new step 0.001 seconds in advance. For example, before 1, I add the time-step 0.999. I ran the RK4 algorithm at various time-steps with the additional steps, and the results were improved significantly, as can be seen in Figure 4.

Exercise 2

(a) The solution operates by first taking $tanh$ of the sensor values. This ensure that the values returned are within a reasonable range below 1. I set forward velocity equal to $1.5 \cdot tanh(front - 0.5)^5$, which means the robot will reverse if it finds itself too close to a wall as $tanh(-0.5) < 0$. Additionally, I change the angular velocity based on the values of front left and front right. I get the robot to move more right as the left sensor get closer and visa versa for the right sensor. See Figure 5 for a plot of it's trajectory.

(b) The rule based controller moves forwards at 1 m/s, until the front sensor detects it is within 0.8m of the wall, when it reverses at 0.5 m/s. This is

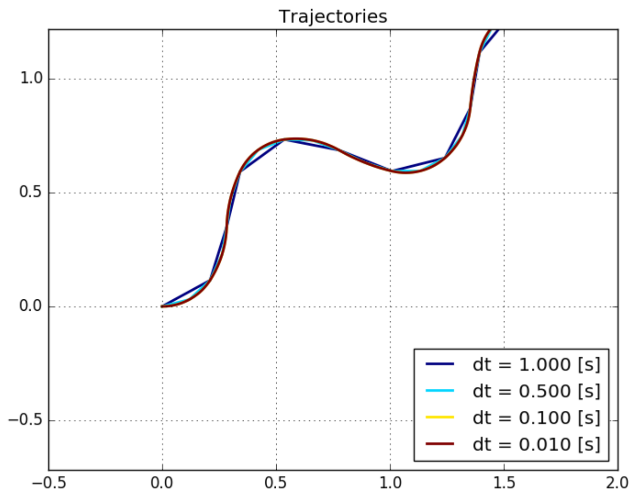


Figure 4: RK4 Trajectories with Dynamic Step-Size

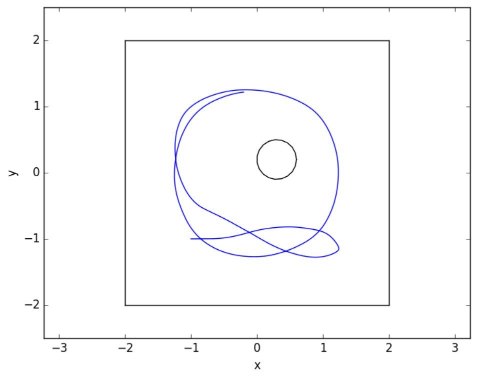


Figure 5: Trajectory of Braitenberg Controller

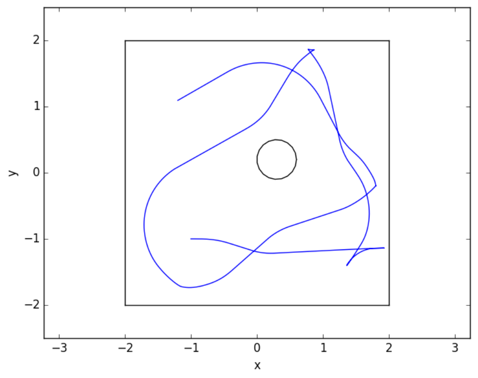


Figure 6: Trajectory of Rule Based Controller

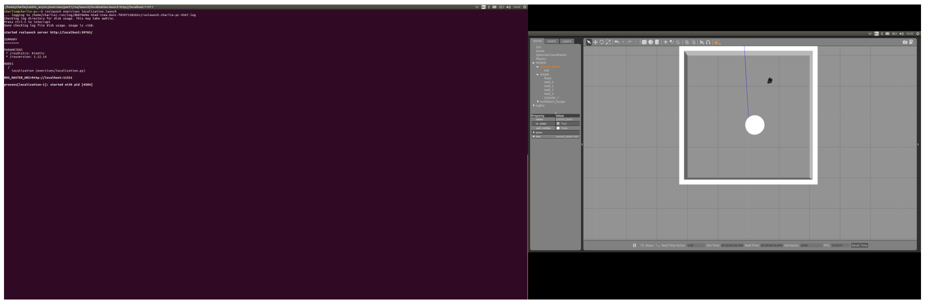
to allow it to back away from walls. Additionally, if the front left sensor is within 1m of the wall, the robot will turn right with angular velocity 1 m/s. Similarly, if the front right sensor is within 1m of the wall, the robot will turn left with angular velocity 1 m/s. See Figure 6 for a plot of it's trajectory.

(c) Having added obstacles to the course, I can draw some comparisons between the methods. My Braitenberg controller would occasionally come to a halt, when a wall was placed a certain distance away. This is as $tanh(0) = 0$, and hence at approximately 0.5m from the wall, my robot would pause. I found that often due to noise in the sensor readings, the robot would overcome this as the sensor readings were never stable at 0.5. I found despite this, the robot was very good at avoiding obstacles even when placed directly in front of it, while in motion. One situation the Braitenberg controller did not deal well with was when placed in a narrow corridor as it does not try to turn or to reverse out.

The rule-based controller is good as it can be programmed for different situations. For example, in the case it comes into a narrow corridor, we can make it turn around. However, we are unable to predict every situation and hence cannot ensure the reliability of a rule-based controller in all layouts.

(d) Without noise, we find it is more likely the Braitenberg controller gets the robot stuck. This is as the robot can enter a point 0.5 m from a wall where the forward velocity remains constant. Additionally, noise can help with turning, as the sensor readings without noise may create a 'tie' between the left and right sensors, resulting in the robot not turning. Random noise can break this tie.

With a rule-based approach we are more able to prepare for scenarios with exact sensor readings. With noise, a rule based approach can appear



```
/home/charlie/catkin_ws/src/exercises/part1/ros/launch/localization.launch http://localhost:11311
charlie@charlie-pc:~$ roslaunch exercises localization.launch
... logging to /home/charlie/.ros/log/db87de0a-42ad-11ea-8a1c-f8597158262c/roslaunch-charlie-pc-4567.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost:39765/

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES
  /
    localization (exercises/localization.py)

ROS_MASTER_URI=http://localhost:11311

process[localization-1]: started with pid [4584]
```

Figure 7: Setup of localization exercise

erratic, as it may switch between cases of an if-statement rapidly. Whereas without noise we can be certain any conditionals are true.

Exercise 3

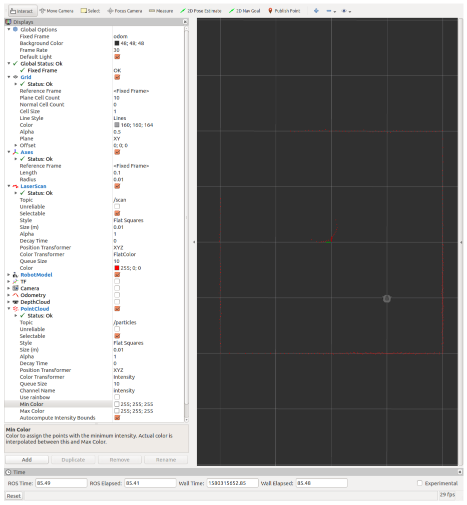
(a) See Figure 7.

(b) Initially, each particle is assigned a random position within the walls. This is done by getting a random float in the range $[-2, 2)$. To ensure that the robot is not within the cylinder, I set up a while loop such that while the position is not valid, it will select another position randomly. The `is_valid` subroutine is implemented such that the particle must be outside the cylinder but within the walls.

(c) To calculate the next position of the particle, I take `delta_pose`, which contains the forward displacement as `X`, and the change in heading as `YAW`. I take these values, and multiply by a random number in the Gaussian distribution with normal 1 and standard deviation 0.1. This ensures the particles move differently, allowing them to deviate. Then, I add the components of the displacement to the current position, and rotate by the change in heading. Figure 8 shows my setup, Figure 9 shows my particles.

(d) To calculate the weight of a particle, first a ray trace runs for each angle from left to right, to provide an accurate measure of how far the particle is from a wall. Values of the actual sensor readings and ray-traced values are both clipped to 3.5 m as this is the maximum range of the sensors. Then, errors are computed at each angle, by inputting the ray-traced distance of the particle into a Gaussian probability density function with mean of the actual sensor reading (from the robot), and standard deviation of 80 cm. Finally the errors are combined by computing the average error, and weights are set to 0.05 if the particle position isn't valid.

Initially, errors were combined by producing a product of them, however this resulted in behaviour that occasionally converged on an incorrect solution. This was as taking the product produced a distribution of weights



Interact | Move Camera | Select | Focus Camera | Measure | 2D Pose Estimate | 2D Nav Goal | Publish Point

Displays

Global Options	
Fixed Frame	odom
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
✓ Status: Ok	
Reference Frame	<Fixed Frame>
Plane Cell Count	10
Normal Cell Count	0
Cell Size	1
Line Style	Lines
Color	▨ 160; 160; 164
Alpha	0.5
Plane	XY
Offset	0; 0; 0
Axes	✓
✓ Status: Ok	
Reference Frame	<Fixed Frame>
Length	0.1
Radius	0.01
LaserScan	✓
✓ Status: Ok	
Topic	/scan
Unreliable	☐
Selectable	✓
Style	Flat Squares
Size (m)	0.01
Alpha	1
Decay Time	0
Position Transformer	XYZ
Color Transformer	FlatColor
Queue Size	10
Color	🟥 255; 0; 0
RobotModel	✓
TF	☐
Camera	☐
Odometry	☐
DepthCloud	☐
PointCloud	✓
✓ Status: Ok	
Topic	/particles
Unreliable	☐
Selectable	✓
Style	Flat Squares
Size (m)	0.01
Alpha	1
Decay Time	0
Position Transformer	XYZ
Color Transformer	Intensity
Queue Size	10
Channel Name	intensity
Use rainbow	☐
Min Color	☐ 255; 255; 255
Max Color	☐ 255; 255; 255
Autocompute Intensity Bounds	✓

Min Color
Color to assign the points with the minimum intensity. Actual color is interpolated between this and Max Color.

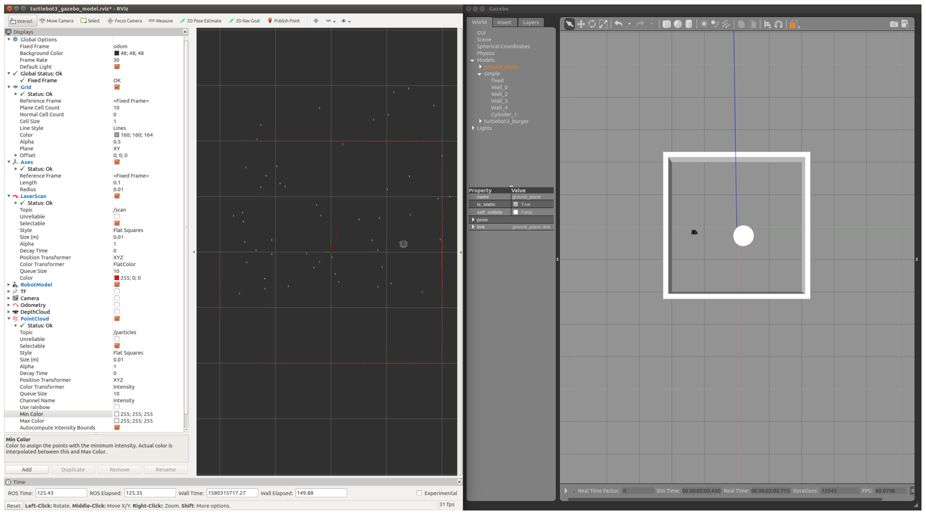
Add | Duplicate | Remove | Rename

Time

ROS Time: 85.49 ROS Elapsed: 85.41 Wall Time: 1580315652.85 Wall Elapsed: 85.48 ☐ Experimental

Reset 29 fps

Figure 8: Setup of RViz



turtlebot3_gazebo_model.rviz* - RViz

Interact | Move Camera | Select | Focus Camera | Measure | 2D Pose Estimate | 2D Nav Goal | Publish Point

Displays

Global Options	
Fixed Frame	odom
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
✓ Status: Ok	
Reference Frame	<Fixed Frame>
Plane Cell Count	10
Normal Cell Count	0
Cell Size	1
Line Style	Lines
Color	▨ 160; 160; 164
Alpha	0.5
Plane	XY
Offset	0; 0; 0
Axes	✓
✓ Status: Ok	
Reference Frame	<Fixed Frame>
Length	0.1
Radius	0.01
LaserScan	✓
✓ Status: Ok	
Topic	/scan
Unreliable	☐
Selectable	✓
Style	Flat Squares
Size (m)	0.01
Alpha	1
Decay Time	0
Position Transformer	XYZ
Color Transformer	FlatColor
Queue Size	10
Color	🟥 255; 0; 0
RobotModel	✓
TF	☐
Camera	☐
Odometry	☐
DepthCloud	☐
PointCloud	✓
✓ Status: Ok	
Topic	/particles
Unreliable	☐
Selectable	✓
Style	Flat Squares
Size (m)	0.01
Alpha	1
Decay Time	0
Position Transformer	XYZ
Color Transformer	Intensity
Queue Size	10
Channel Name	intensity
Use rainbow	☐
Min Color	☐ 255; 255; 255
Max Color	☐ 255; 255; 255
Autocompute Intensity Bounds	✓

Min Color
Color to assign the points with the minimum intensity. Actual color is interpolated between this and Max Color.

Add | Duplicate | Remove | Rename

Time

ROS Time: 125.43 ROS Elapsed: 125.35 Wall Time: 1580315717.27 Wall Elapsed: 149.88 ☐ Experimental

Reset **Left-Click:** Rotate. **Middle-Click:** Move X/Y. **Right-Click::** Zoom. **Shift**: More options. 31 fps

Figure 9: Particles displayed in RViz

where small errors multiply, so there would be many weights around 0, meaning all the particles quickly converge on a higher weight. Hence, I changed to averaging the error which produces a distribution of weights with more variation.

The weights are set to 0.05 if the particle position isn't valid, in order to attempt to relocate any particles that the robot cannot physically be near. Initially, the weights were set to some multiple of their computed errors if they were out of bounds. However this was not a good solution, as once a particle is out of bounds, the computed errors tell us nothing since the robot could never be there.

(e) Occasionally unsuccessful localization happens as a result of the symmetry in the scene - the particles can get similar sensor readings from a location that is different to the current robot's position.

Convergence is often slow when the robot is moving slowly, as the particles themselves will vary their motion less, meaning they take more time to randomly locate the correct position.

(f) At present, when the robot is kidnapped, the particles do not move with it. This is as they have already converged on where they think the robot is, so random motion alone will not find the new position. Instead, we must assign each particle a small probability p of being randomly relocated to anywhere within the scene. This random relocation covers the cases where the robot is moved.

We can change the value of p to tune the performance - a higher value of p will relocate the particles more - meaning it will very quickly discover the robot after it had been relocated. However we often desire a lower value of p as otherwise the random displacement of particles will start to increase the error.

Figure 10 shows error with two kidnapping attempts, using a value $p = 0.005$. The first at about timestep 250, and the second at 500. This shows a good compromise between low error once it has converged and quick convergence even after kidnapping, as can be seen in the figure.

(g) Trajectory and localization error are displayed in Figure 11. This particle filter performs quite well, it converges within 300 timesteps, and once it has, error does not rise above 30 cm.

(h) The extended Kalman Filter estimates current position based off of Gaussian distributions across the state space. It has the benefit in this case of easily dealing with Guassian measurement noise, as it assumes measurement noise within it's calculations. Additional benefits of a Kalman Filter over the particle filter include memory and time efficiency and greater resolution.

However, the Kalman filter would be inappropriate in our situation - it requires a known initial pose which we do not have. Additionally, it could

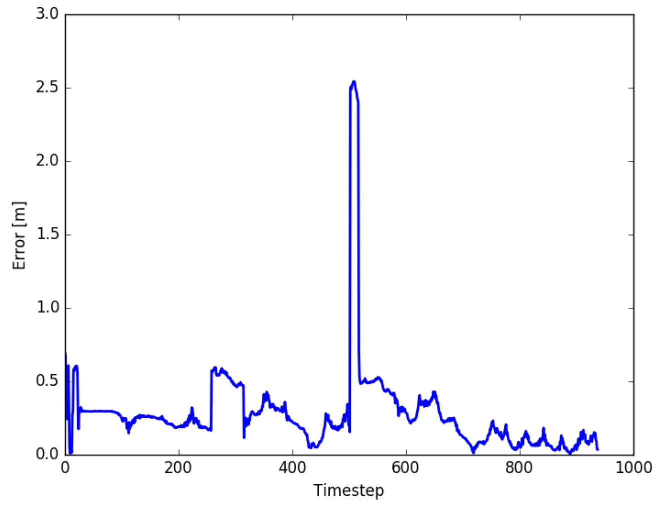
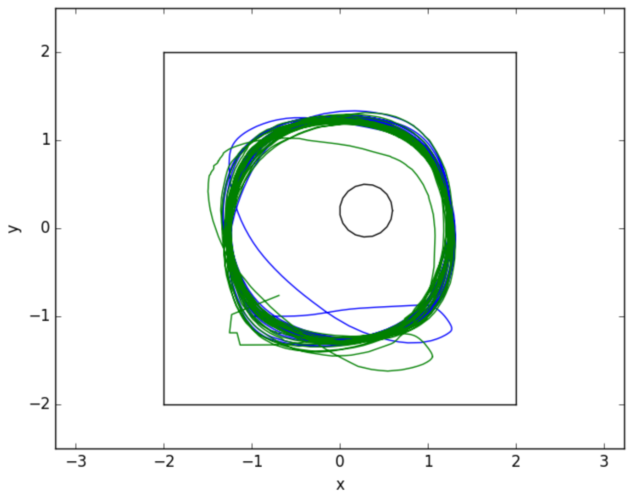
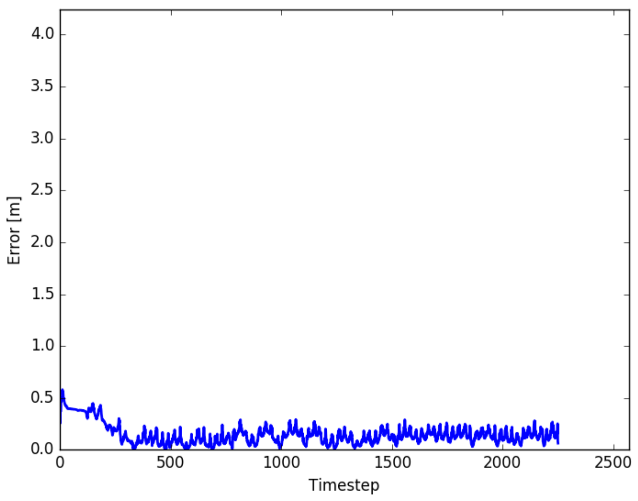


Figure 10: Error with Kidnapping

not deal with the kidnapping problem as it relies on the movement model to predict the next state, and does not provide provisions for random movement we cannot detect.



(a) Trajectory



(b) Error

Figure 11: Particle Filter Performance