Mobile Robotic System Practical 1

Elim Yi Lam Kwan (ylk25)

1 Exercise 1

1.1 Part (a)

The differential equations that correspond to the motion of the differential wheeled robot are:

$$\dot{x} = u \cos\theta \tag{1}$$
$$\dot{y} = u \sin\theta \tag{2}$$
$$\dot{\theta} = \omega \tag{3}$$

where u, ω, t represents the forward velocity, the rotational velocity and time, while x, y refers to the position coordinates on the 2D plane. θ is the angle between the horizontal axis and the forward velocity. Moreover, the ground truth of the robot trajectories was given as $\omega = \cos t$, hence, one would expect a periodic cosine motion path in later experiments of Section 1.

1.2 Part (b)

Euler method could be used to update the robot pose. The principle behind is that given a robot's starting point (x, y, θ), the slope$(\dot{x}, \dot{y}, \dot{\theta})$ at that point can be computed with the differential equations in Section 1.1. The next point can then be estimated by taking a small step along the tangent line. The general formula is:

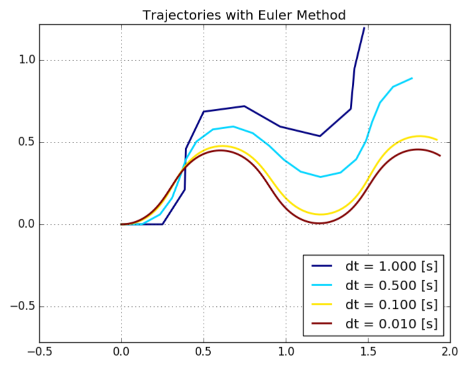
$$y_{n+1} = y_n + dtf(t_n, y_n) \tag{4}$$
$$\tag{5}$$

where y_{n+1} is our updated robot pose estimated from the existing pose y_n, step size would be the changes in time dt, and the function f is the ordinary differential equations listed in Equation 1, 2, 3.

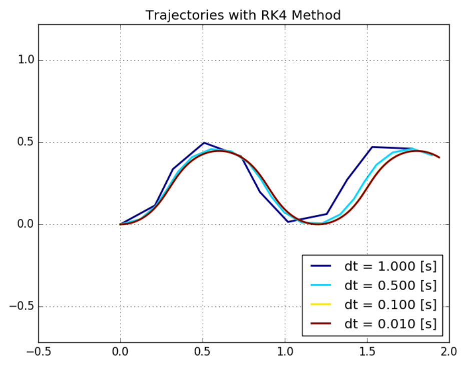
1.3 Part (c)

Figure 1a shows the estimated trajectories with the Euler method. We observed that at small dt ($dt = 0.01$), the estimated trajectories was similar to the expected cosine curve, which was not the case for other step size settings. At $t = 0.4$, trajectories associated with $dt \geq 0.1$ begin to diverge. Moreover, the errors were more significant when a large step size was adopted. This can be explained as the Euler method is a first-order method and errors are accumulative since there is no PID control in place for error correction. The error per step is proportional to the square of the step size and the error at a given time is proportional to the step size, which explained the phenomenon observed. Hence, a small resultant error is the main advantage for adopting a small step size.

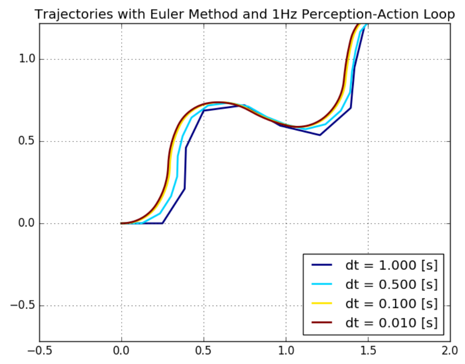
However, the draw-back of using a small step size is computational requirements. Small step size implies that we are inquiring the sensors' more frequently, which increases the system's power consumption.



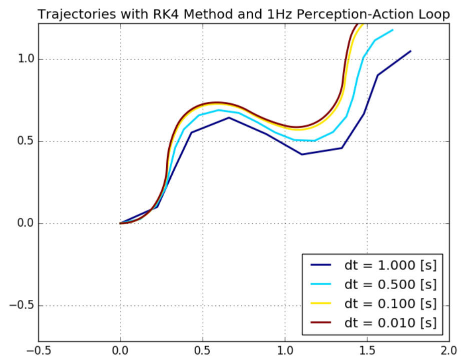
(a) Euler Method



(b) RK4 Method



(c) Euler with 1Hz Perception-Action Loop



(d) RK4 with 1Hz Perception-Action Loop

1.4 Part (d)

Alternatively, Runge–Kutta methods (RK4) is a fourth order method for robot pose estimation. RK4 determines the next value with the weighted average of the slopes within the time interval and the present value. It is written as:

$$y_{n+1} = y_n + \frac{1}{6}dt(k_1 + 2k_2 + 2k_3 + k_4) \tag{6}$$

$$\text{where, } k_1 = f(t_n, y_n) \tag{7}$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}) \tag{8}$$

$$k_3 = f(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}) \tag{9}$$

$$k_4 = f(t_n + h, y_n + hk_3) \tag{10}$$

y_n, y_{n+1} and $f()$ were defined in the Euler equations. Given an interval from t to $t + dt$, k_1 and k_4 corresponds to the slopes at the beginning and the end of the interval, while k_2 and k_3 corresponds to the slopes at the midpoint of the interval. Note that the computation of k_1 resembles the Euler method.

Also, we took into consideration that $\dot{x}, \dot{y}$ depend on θ, where θ is also proportional to time t. Thus, when computing k_{2-4} for the x, y coordinates, $\theta_{t_n+\frac{h}{2}}$ and θ_{t_n+h} have been calculated through intrapolation:

$$\theta_{t_n+\frac{h}{2}} = \theta_{t_n} + \frac{1}{2}cos(\theta_{t_n})dt \tag{11}$$

$$\theta_{t_n+h} = \theta_{t_n} + cos(\theta_{t_n})dt \tag{12}$$

1.5 Part (e)

When comparing the Euler and RK4 methods in Figure 1a and 1b, we observed that small step sizes, such as $dt = 0.01$, generated trajectories that are very close to the ground truth cosine curve pattern. The difference between RK4 and Euler lies in their performance with larger step sizes. The estimated curves diverged from the expected values much more quickly in the case of Euler than in RK4. These behaviours can be explained by the order of the methods, and its implications on the truncation and accumulated errors. Since RK4 is a fourth-order method, local truncation and accumulated error are in the order of $O(dt^5)$ and $O(dt^4)$ only, which means it has a much slower growth rate of errors compared with the first order method, Euler.

Moreover, if the system dynamics are slow, computing four tangent lines may be wasteful because the changes in slopes are small. In those scenarios, the simple Euler method can also provide accurate results while being less computational expensive, and hence is preferred.

1.6 Part (f)

By truncating the t in the $w = \cos(t)$ function, we can emulate the effect of a perception-action loop running at 1Hz. Its impact was shown in Figure 1c and 1d. Trajectories from both schemes diverged from the expected cosine graph since the first peak. This happened because turning angle errors accumulates when t is not equal to an integer. When t is not an integer point, the true angle update should be $cos(t)$ instead of $\cos(\text{floor}(t))$. The delayed angle updates caused the agent to drift apart from its expected values, even though it periodically turned in the right directions when t is an integer, it is unable to correct its previous errors. In an open-loop system, errors will continue to manifest. In the case of RK4, $dt = 1$ shows the least amount of errors because its update frequency is closest to the frequency of the perception-action loop. Thus, it has fewer truncation errors. This phenomenon is less visible with the Euler method because it already accumulated a lot of errors even without the perception-action loop when dt is large.

1.7 Part (g)

Fixed step size might be inefficient in some cases. An adaptive scheme would allow us to increase the step size when error is small. Vice versa. On top of that, adaptive step size might also help reduce errors. To determined the truncation error caused by the perception-action loop per step, Runge–Kutta–Fehlberg method (RKF45) could be used. The idea behind is that predicted robot pose can be estimated by two methods with different order, and in our case RK5 and RK4. Then, the errors were approximated as the difference between the predicted output from RK5 y_{n+1} and RK4 $y_{n+1}*$:

$$e_{n+1} = y_{n+1} - y_{n+1}* \tag{13}$$

With the calculated error, we could reassign the step size to a more desired value and re-compute the current predictions with a smaller step size when error is larger than tolerance. Our step size dt was updated with the logic below:

Algorithm 0: RKF54 Logic

```
error, next_pose = Prediction(current_pose, time, dt)
if $error > tolerance$ then
    dt = 0.8*dt;
    Repeat Prediction(current_pose, time, dt);
else
    dt = 1.2*dt;
```

Moreover, given that the fixed perception-action loop takes 1 second, the maximum step size was set to 1, such that no information would be lost. Figure 2 demonstrated the performance gain in adopting RKF45 to handle truncation error. The effectiveness of RKF45 could be shown by comparing Figure 2 against Figure 1d. It shows that RKF45 achieved better results (a smoother curve with reduced errors) with much fewer samples than RK4. For example, RK4 could achieve similar error performance with $dt = 1$, but the curve was not smooth, while the smooth curve generated by RK4 required 100 samples, which doubled the amount of that in RKF45 and it had more errors as well. Lastly, as potential extensions, Euler could also be paired with Heun's method using a similar methodology to alleviate the truncation error observed in Figure 1c.

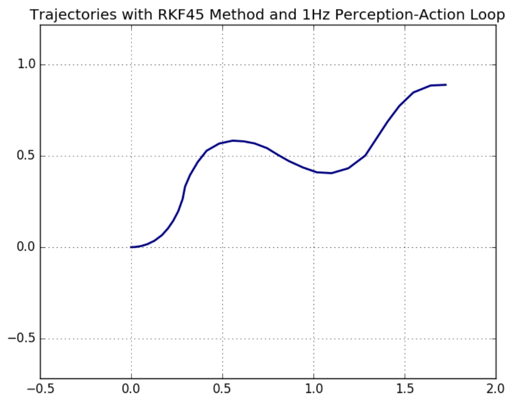


Figure 2: RKF45 with 1Hz Perception-Action Loop and Adaptive Step Size (Total number of sampled is 42 in a 10 seconds period)

2 Exercise 2

2.1 Part (a)

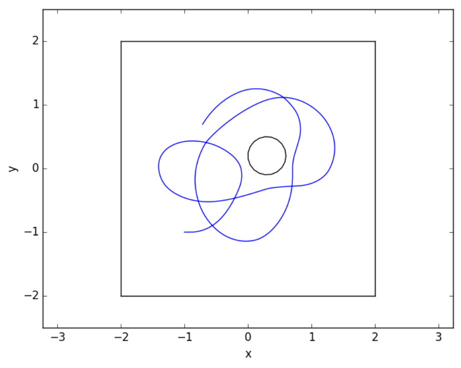
Braitenberg vehicle refers to the concept that sensors' inputs are linked to the control output with minimal logic, such that the agent can move around autonomously with basic reactive behaviours. Below shows our implementation of the Braitenberg controller:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} w_1 & w_2 & w_3 & w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 & w_{10} & w_{11} & w_{12} \end{bmatrix} \text{Sigmoid}\left(\begin{bmatrix} \text{front} \\ \text{front}_{\,left} \\ \text{front}_{\,right} \\ \text{left} \\ \text{right} \\ \text{front} * \text{left} \end{bmatrix} \right) \tag{14}$$

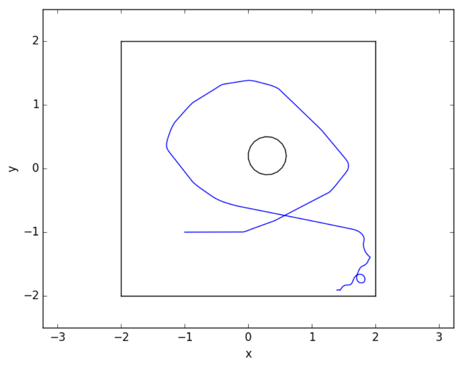
$$\text{where the weights } w_i \text{ were tuned as } \begin{bmatrix} 0.3 & 0.1 & 0.1 & 0 & 0 & 0 \\ 0 & 1.1 & -1.1 & 0.9 & -0.9 & 0.5 \end{bmatrix}$$

Sigmoid: Since the raw sensor input will be '$infinte$' when obstacles are out of the range of the sensors, a Sigmoid Function with offset was applied to clip the input. **Forward Velocity:** It is desired that the speed of the agent will reduce when it approaches obstacles. Hence, the front sensors' weights w_{1-3} were positive, with heavier weightings for the more dominant middle sensor. **Angular Velocity:** The turning angle of the agent was determined by the relative difference between the left and right sensors, which explains w_{8-11}. **Edge Case:** Since the differences in left and right sensors will not indicate head-on collision scenarios, the additional $(front * left)$ term was added. It increases the turning angle when there is an obstacle ahead, with the $(left)$ term used to regulate the magnitude of the angle and reduce the variable dependence on $(front)$. w_7 was not used for this purpose, because when the agent approached the wall, it requires a large turning angle. However, if w_7 was set too high, which means the robot would keep on turning in circles when the front sensor is cleared and this is undesirable.

Figure 3a shows the trajectory of the agent. It can be observed the motion is quite smooth and it changes directions gradually.



(a) Braitenberg Controller



(b) Rule-based Controller

2.2 Part (b)

For the rule-based controller, four major rules were developed, as outlined in Algorithm 1.

Algorithm 1: Rule-based Controller

```
if Head-on Collision or Glued then
    Slow down. Turn roughly $180^\circ$ around;
else if Left Obstacle then
    Slow down. Turn Right by $(\text{diff}_{\text{left,right}} + 0.2 * e^{\text{diff}_{\text{left,right}}})$ ;
else if Right Obstacle then
    Slow down. Turn Left by $(\text{diff}_{\text{left,right}} + 0.2 * e^{\text{diff}_{\text{left,right}}})$ ;
else
    No turning. Keep walking forward;
```

The agent exploration at a given time step can be generally concluded as encountering obstacles ahead, encountering obstacles on its left/ right side, nothing happening, and something going wrong. For the first three cases, the agent should steer in the opposite direction of the obstacle. The exponential term in the equation increases the turning angle significantly when the agent is close to the obstacles. For the latter two cases, the agent should either continue to move forward or turn 180° around because it is likely that it entered a dead end. The agent will also slow down during turning to minimise collisions.

Figure 3b shows the trajectory of the agent. There were more sharp turns and the path was less smooth compared with that of the Braitenberg controller. Its ability in avoiding obstacles also seem to be inferior to that of Braitenberg, as it was bumping into walls at the lower right corner. This can be explained as it is quite difficult to cover all the possible scenarios encountered by the robot with the rule-based controller. A possible solution would be to develop a more comprehensive set of rules, however, the resultant controller would have higher complexity than that of Braitenberg.

2.3 Part (c)

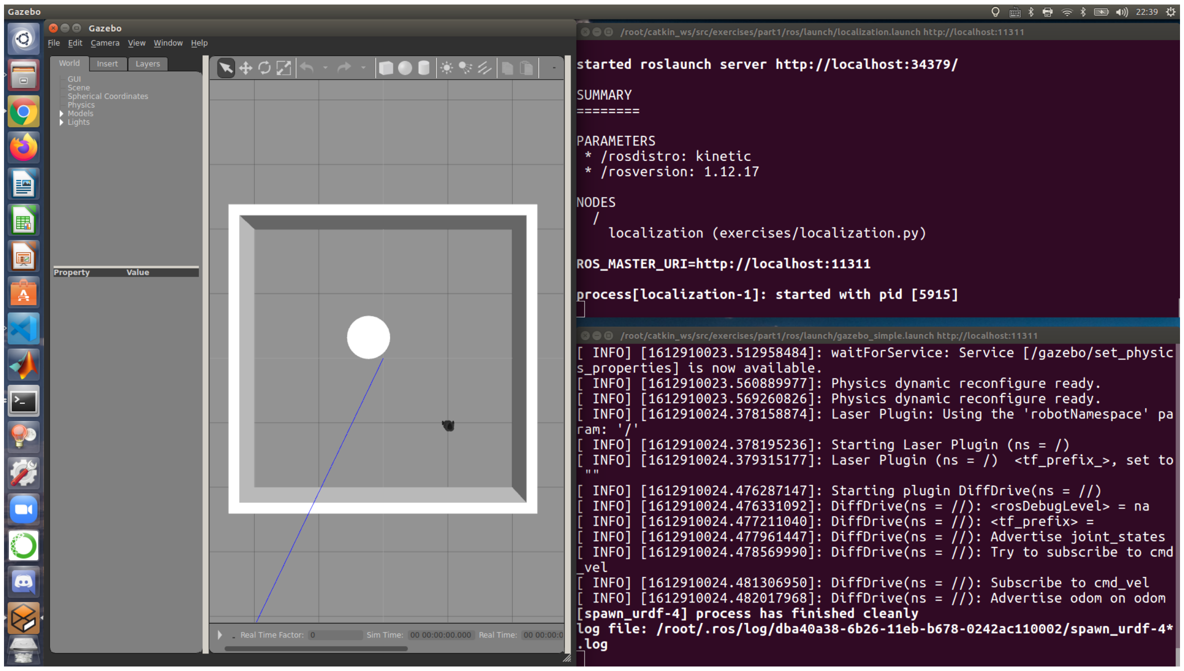
In terms of robustness, the controllers were evaluated based on their performance in face of obstacles with different shapes, narrow corridors and suddenly appears of obstacles. Both controllers could tackle cube, cylinder and sphere of reasonable sizes. Braitenberg controller is more superior in terms of exploring new places and escaping from narrow spaces, whereas the rule-based controller can react more swiftly to obstacles that suddenly emerge in front of it. This can be explained as the rule-based controller can make sharp turns to steer away, while Braitenberg is unable to do so. Therefore, in terms of robustness in handling dynamic obstacles, the rule-based controller is more superior.

2.4 Part (d)

Currently, the agents were simulated with random sensors' noise. In the case of Braitenberg controllers, the control outputs are directly proportional to the sensor's inputs, these additional noises will cause the agent to turn, even when there are no obstacles. In the case of rule-based controller, if noises are large enough to trigger the conditions, it will also cause unpredicted movements of the agent. The advantage of perfect sensors is that it can eliminate the above errors. However, sensor noises are unavoidable in the physical world, and shielding it in our simulation will lead to a solution that is much less robust to real-world variances.

3 Exercise 3

3.1 Part (a)



```
/root/catkin_ws/src/exercises/part1/ros/launch/localization.launch http://localhost:11311
started roslaunch server http://localhost:34379/

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.17

NODES
  /
    localization (exercises/localization.py)

ROS_MASTER_URI=http://localhost:11311

process[localization-1]: started with pid [5915]
```

```
/root/catkin_ws/src/exercises/part1/ros/launch/gazebo_simple.launch http://localhost:11311
[ INFO] [1612910023.512958484]: waitForService: Service [/gazebo/set_physic
s_properties] is now available.
[ INFO] [1612910023.560889977]: Physics dynamic reconfigure ready.
[ INFO] [1612910023.569260826]: Physics dynamic reconfigure ready.
[ INFO] [1612910024.378158874]: Laser Plugin: Using the 'robotNamespace' pa
ram: '/'
[ INFO] [1612910024.378195236]: Starting Laser Plugin (ns = /)
[ INFO] [1612910024.379315177]: Laser Plugin (ns = /)  <tf_prefix_>, set to
 ""
[ INFO] [1612910024.476287147]: Starting plugin DiffDrive(ns = //)
[ INFO] [1612910024.476331092]: DiffDrive(ns = //): <rosDebugLevel> = na
[ INFO] [1612910024.477211040]: DiffDrive(ns = //): <tf_prefix> =
[ INFO] [1612910024.477961447]: DiffDrive(ns = //): Advertise joint_states
[ INFO] [1612910024.478569990]: DiffDrive(ns = //): Try to subscribe to cmd
_vel
[ INFO] [1612910024.481306950]: DiffDrive(ns = //): Subscribe to cmd_vel
[ INFO] [1612910024.482017968]: DiffDrive(ns = //): Advertise odom on odom
[spawn_urdf-4] process has finished cleanly
log file: /root/.ros/log/dba40a38-6b26-11eb-b678-0242ac110002/spawn_urdf-4*
.log
```

Figure 4: Setup Image for Exercise 3(a)

Figure 4 demonstrated the setup of the experiment. The top and bottom terminals correspond to the ones that were used to run the controller and the Gazebo software. The Braitenberg controller developed in the previous section shows adequate performance, hence, it was reused in this Section.

3.2 Part (b)

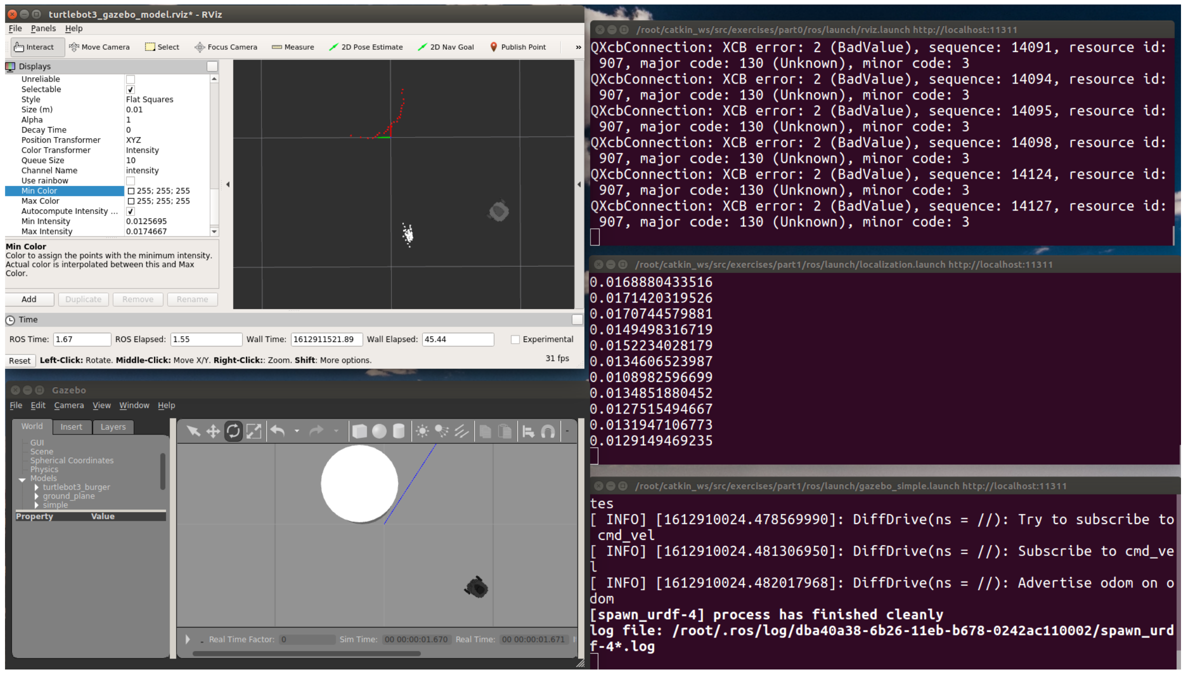
Functions $random\_pose()$ and $is\_valid()$ were implemented to assist the initialisation of particles, where we would like to randomly place particles within the arena. $random\_pose()$ samples 3 decimal numbers from the uniform distribution to assign it to the robot pose (x, y, ω), whereas $is\_valid()$ checks if the generated pose is valid, taking into consideration the location of the wall, the size of the cylinder obstacle and the robot dimension. During the initialisation process, random poses are generated until a valid one is found.

3.3 Part (c)

The setup with RViz is shown in Figure5. On the top left, it illustrated the obstacles points (in red) and the particles (in white). In $Particle.move()$, poses were updated according to the motion model. The motion model are Gaussian distributions centred around the forward velocity u and the rotational velocity w, with standard deviations sd being 10% of u and w respectively:

$$u' = \mathcal{N}(u, sd) \qquad w' = \mathcal{N}(w, sd) \tag{15}$$

The updated forward and rotational velocity would be random numbers drawn from the distribution listed in Equation 15 and they could be translated to world coordinates with Equation 1-3.


turtlebot3_gazebo_model.rviz* - RViz
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point
Displays
Add Duplicate Remove Rename
Time
Reset
Experimental
31 fps
Gazebo
File Edit Camera View Window Help
World Insert Layers
GUI
Scene
Spherical Coordinates
Physics
Models
turtlebot3_burger
ground_plane
simple
Property Value
Real Time Factor: 0
Sim Time: 00 00:00:01.670
Real Time: 00 00:00:01.671

```
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 ...   ✓
Min Intensity               0.0125695
Max Intensity               0.0174667
```

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

ROS Time: 1.67 ROS Elapsed: 1.55 Wall Time: 1612911521.89 Wall Elapsed: 45.44

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

/root/catkin_ws/src/exercises/part0/ros/launch/rviz.launch http://localhost:11311

```
QXcbConnection: XCB error: 2 (BadValue), sequence: 14091, resource id:
 907, major code: 130 (Unknown), minor code: 3
QXcbConnection: XCB error: 2 (BadValue), sequence: 14094, resource id:
 907, major code: 130 (Unknown), minor code: 3
QXcbConnection: XCB error: 2 (BadValue), sequence: 14095, resource id:
 907, major code: 130 (Unknown), minor code: 3
QXcbConnection: XCB error: 2 (BadValue), sequence: 14098, resource id:
 907, major code: 130 (Unknown), minor code: 3
QXcbConnection: XCB error: 2 (BadValue), sequence: 14124, resource id:
 907, major code: 130 (Unknown), minor code: 3
QXcbConnection: XCB error: 2 (BadValue), sequence: 14127, resource id:
 907, major code: 130 (Unknown), minor code: 3
```

/root/catkin_ws/src/exercises/part1/ros/launch/localization.launch http://localhost:11311

```
0.0168880433516
0.0171420319526
0.0170744579881
0.0149498316719
0.0152234028179
0.0134606523987
0.0108982596699
0.0134851880452
0.0127515494667
0.0131947106773
0.0129149469235
```

/root/catkin_ws/src/exercises/part1/ros/launch/gazebo_simple.launch http://localhost:11311

```
tes
[ INFO] [1612910024.478569990]: DiffDrive(ns = //): Try to subscribe to
 cmd_vel
[ INFO] [1612910024.481306950]: DiffDrive(ns = //): Subscribe to cmd_ve
l
[ INFO] [1612910024.482017968]: DiffDrive(ns = //): Advertise odom on o
dom
[spawn_urdf-4] process has finished cleanly
log file: /root/.ros/log/dba40a38-6b26-11eb-b678-0242ac110002/spawn_urd
f-4*.log
```

Figure 5: Setup Image for Exercise 3(c)

Finally, to handle the kidnapped robot problem, we would like to increase the exploration conducted by the particles by re-positioning them from time to time. Also, the re-positioning event should only occur with low probabilities, such that the model will still convergence. In our implementation, a random variable was drawn from the uniform distribution between 0 to 1, if its value is below a certain threshold, the particles' pose will be generated by the $random\_pose()$ function instead of being updated by the motion model stated above.

3.4 Part (d)

For a particle, we can obtain the distance z_t from each of its sensors to its nearest obstacles with the $ray\_trace()$ function. A measurement model with standard deviation of $\sigma = 0.8$ could then be generated:

$$\mathcal{N}(z_t, \sigma^2) \tag{16}$$

Afterwards, to obtained the probability of the particle pose being the actual pose of the robot given the sensors data x_t measured, we could evaluate the probability distribution function in Equation 16 at the sensor values measured, which equate to:

$$p(z_t|x_t) = \mathcal{N}(z_t, \sigma^2).pdf(x_t) \tag{17}$$

Moreover, particles with invalid pose were also penalised by reducing their weights by half. **Tuning:** In addition, particle weight may collapse when a few ensemble members receive most of the weight, leaving other particles with little weights. Hence, the parameters σ^2 in the measurement model (Equation 16 and 17) was increased to de-emphasizing small-scale differences between the particles. Also, in the motion model, the pose was only updated when the changes were large enough. Furthermore, we noticed that increasing the sd in the motion model (Equation 15) also help convergence.

3.5 Part (e)

Upon testing, localization usually succeed if there were enough time to converge and that the vehicle did not get caught up by obstacles. We noticed that when a few of the randomly initialised particles meet up with the robot, their motion paths will quickly converge. Occasionally, when the robot was roaming around the cylinder, the particle cluster converged to the wrong location and ended up lagging behind the robot. This could be explained as our scene set up is quite symmetric, the model ended up believing different corners of the room being identical places.

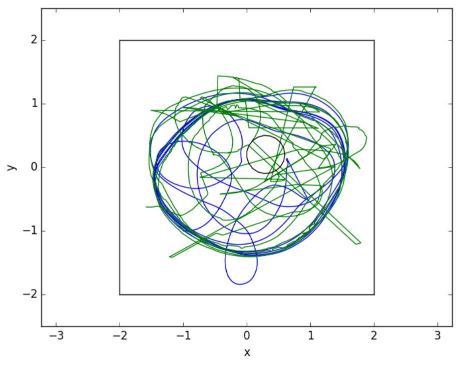
3.6 Part (f)

In the case of the kidnapped robot, the localization model converged quickly if the robot had not been carried away too far. If the robot pose deviated a lot, the localization model would still be able to re-track the robot, however, it might take longer period of time. This is because it relied on the robot to collide with the particles which were initialised at random positions. To improve the model robustness in terms of handling the kidnapped robot case, the followings parameters in the motion model were tuned:

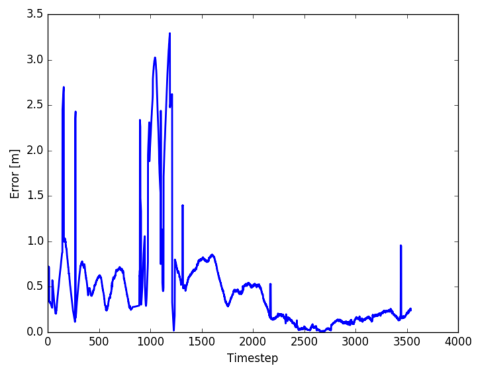
- $reposition\_threshold$: It governs the probability of the occurrence of particle re-position. It was increased.
- sd: It controls the variances in the motion model. It was increased.
- $update\_threshold$: It controls whether the particle pose is being updated. If the changes proposed is below the threshold, the update will not occur. The parameter was reduced.

The above measures added variance to the model and enhanced its exploration ability, hence, improved its robustness.

3.7 Part (g)

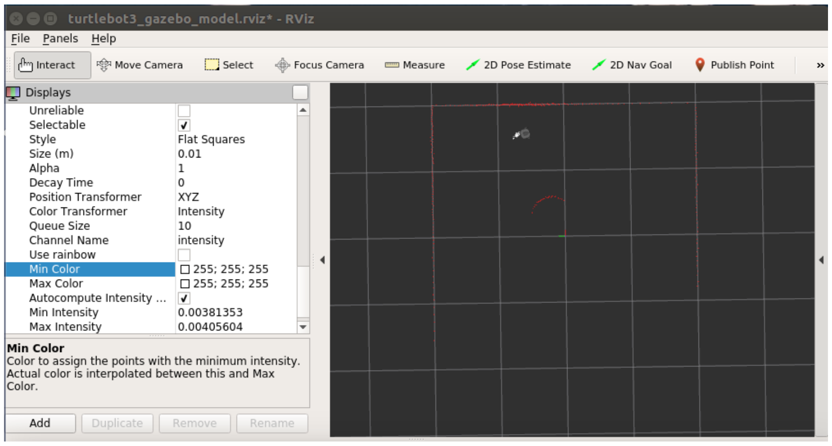


(a) Robot (blue) and Particle (green) Trajectories



(b) Localization Errors

The trajectories of the particle cluster verse the actual location of the robot is as shown in Figure 6a. It can be observed that at the starting phase, the green and blue curves established very different behaviours. This period corresponds to the first 1400 time step period on Figure 6b, where localization error is high. Moreover, the sharp edges on the green line show that particle re-position happened occasionally. Additionally, the error plummeted at around $t = 1400$ and the low localization error implies that the particle cluster and the robot are in synchronisation. Hence, in Figure 6a, there are some overlapping green and blue lines in the shape of an ellipse. Figure 7 shows the point cloud cluster converged around the robot. Time taken to converge may reduce if more particles were used.



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 ...	✓
Min Intensity	0.00381353
Max Intensity	0.00405604

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

Figure 7: Synchronised Point Cloud with Robot

3.8 Part (h)

In this exercise, the localization model often requires 30 to 90 seconds for convergence because the model is waiting for the randomly initialised particles to collide with the robot. Instead of using a Monte Carlo method, Extended Kalman filter (EKF) predicts the robot location based on its prior belief and use measurements to update the posterior. The advantage of EKF over particle filter is that it requires less computational power. The number of points used in a particle filter generally needs to be much greater than the number of points in EKF. In fact, the Kalman filter is the optimal filter for a linear system with Gaussian noise. Moreover, the advantage of particle filter over EKF is that it is more elastic and can handle systems with non-Gaussian noise.