Where Am I - Robot Localization using AMCL

Sandra Schuhmacher

Abstract—A abstract is meant to be a summary of all of the relevant points in your presented work.

Index Terms—Robot, IEEEtran, Udacity, LaTeX, ROS, AMCL, Localization.

1 Introduction

In this project the ROS Navigation Stack in cooperation with the ROS amcl package is used to accurately localize a mobile robot inside a provided map in the Gazebo and RViz simulation environments. First the environment is explored by using a provided robot model, then a custom model is constructed and used.

The goal of this experiment is to learn about the process of creating packages, integrating and utilizing ROS packages into a workspace and simulate them in Gazebo and Rviz. A robot model is created and the parameters of the intergrated packages are tuned until they produce sufficient results.

The robot is localized with the ROS AMCL package and navigates by utilizing the ROS Navigation Stack. For project testing it is provided with a navigation goal in a race-track world, where it has to plan a path around various obstacles to reach its goal.

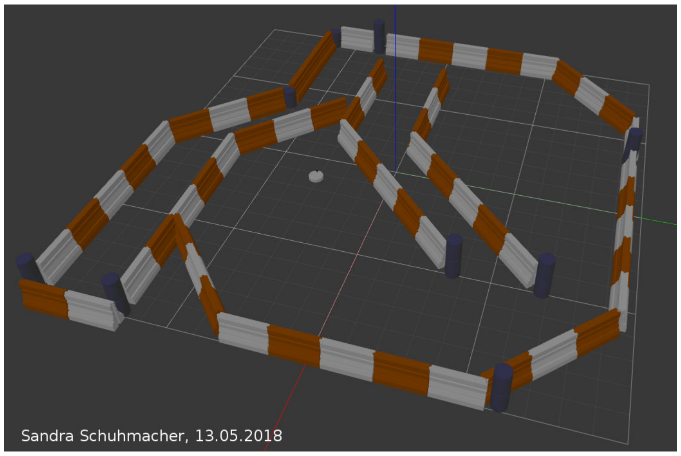


Fig. 1. Robot in the Race World

Using this setup, the problem of localizing a mobile robot is explored. The wheels defined in the urdf have friction and damping defined, which adds slick noise to the odometry. Additionally, the laser sensor in use is beeing distorted with noise so that the sensors do not deliver perfect results. Using the mentioned ROS packages, the robot is able to be localized and navigate to its goal, despite the noise.

2 Background on Localization

A mobile robot faces three major problems in terms of its position:

1) **Position Tracking/Local Localization**
The robot knows its initial pose and the challenge is to keep track of the position when moving through the environment. Those positions have to be filtered from noise from the environment like wheel slippage or interaction with obstacles.
2) **Global Localization**
The robot does not know about its initial pose and the position has to be calculated in relation to the map its operating on. As there is no initial known good pose, the uncertainty of the calculation result is higher then in local localization.
3) **Kidnapped Robot**
The robot operates the same as in global localization, but may be moved to a new and possibly unknown position at any time.

As already mentioned in the local localization, noise plays a major role in localization. A trivial solution to the problem, like reading the robots wheel encoders, fails immediately for two reasons:

- Sensors are never perfectly accurate. Even the most expensive sensors availiable may report wrong measurements. A wheel encoders optical sensor might report too much or too less wheel rotations.
- The robots operation environment impacts the actuators. A wheel might slip on greasy floors or rotate free because its stuck in sand. The sensor correctly reports wheel movement, but it doesn't actually move the robot.

To deal with noise, two approaches are typically used: multiple sensor inputs are used to determine movement and position. This process is called *sensor fusion*. In the project the odometry sensors are fused with a laser range finder. Additionally, a filter is applied to the sensor inputs to reduce the input noise and get a better position estimate by removing the noise uncertainty. Filtering usually appears in two stages: an estimation and an actual measurement. The estimation part of the filtering smoothes out any spikes in the measurements that may occur.

2.1 Localization Algorithms

In robotics there are two popular methods of filtering to determine position: The *Extended Kalman Filter* (EKF) and the *Adaptive Monte Carlo Localization* (AMCL). The difference between the two is briefly covered in 1.

TABLE 1
AMCL vs. EKF

	AMCL	EKF
Measurements	Raw Meas. Input	Landmarks
Meas. Noise Model	Any Distribution	Gaussian Distribution
Posterior Pos. Model	Particles	Gaussian Distribution
State Space	Multimodel Discrete	Unimodal Continuous

While the EKF algorithm is easier to implement, the AMCL algoritm provides better control over memory and resolution by tuning the number of its particles. In case of choosing on over the other, both are valid choices but EKF is unable to solve the global localization problem.

2.2 Extended Kalman Filter

Kalman filters are used to estimate the value of a variable in real time during data collection. By alternating between a *Measurement Update* and a *State Prediction* it is able to determine a very accurate prediction of the variables value after only a few iterations. Kalman Filters can exists in multiple dimensions: whereas a 1D Kalman Filter would estimate a single variable, multidimensional kalman filters are able to estimate multiple dependent variables at once.

2.2.1 Measurement Update

The kalman filter starts its operation with an initial belief about its position. This is the first *Prior Belief*. It then starts with the *Measurement Update*: a gaussian distribution is calculated around the prior belief and the mean and variance of this distribution is extracted. This is repeated for the value of the measurement update.

$$\mu = \text{mean of prior belief}$$

$$\sigma^2 = \text{variance of prior belief}$$

$$v = \text{mean of prior belief}$$

$$r^2 = \text{variance of prior belief}$$

$$EKF \sim \mathcal{N}(\mu, \sigma^2)$$

The new mean which is to be used in the state prediction is a weighted sum of the mean of the prior belief and the mean of the current measurements mean:

$$\mu' = \frac{r^2 \cdot \mu + \sigma^2 \cdot v}{r^2 + \sigma^2}$$

So the result of this calculation is a weighted value between the prior belief and the current measurement. Around this new value, variance for a gaussian is calculated:

$$\sigma^{2'} = \frac{1}{\frac{1}{r^2} + \frac{1}{\sigma^2}}$$

This value is called the *Posterior*.

2.2.2 State Prediction

The *State Prediction* takes place after a movement: the posterior is taken as an input with its mean and variance to become the prior belief of the prediction step. The amount of motion is distributed and the posterior of the state prediction step is calculated by adding mean and variance of motion and prior belief:

$$\text{Posterior Mean} : \mu' = \mu_1 + \mu_2$$

$$\text{Posterior Variance} : \sigma^{2'} = \sigma_1^2 + \sigma_2^2$$

This basic Kalman Filter algoritm performs well for linear value problems in motion and measurement, but fails to apply as soon as the movement gets nonlinear, the state space can no longer be represented as a gaussian distribution. The extended kalman filter works around this problem by aproximating a non-linear input value with a taylor-series. The general process of iterating between state prediction and measurement update stays the same.

2.3 Adaptive Monte Carlo Localization

Monte Carlo Localization uses a distribution of *particles* that represent possible locations to estimate the current location. Its main difference to EKF is, that it can use any distribution model as an input and is not limited to a gaussian. Each particle is a vector of the robots position values and a weight. The weight is the difference between the particles pose and the robots actual pose.

$$\begin{bmatrix} X_{particle} \\ Y_{particle} \\ \theta_{particle} \\ weight \end{bmatrix}$$

The xyθ-coordinates of the robot are always in reference of the global coordinate frame. Initially the particles are randomly spread onto the entire map. AMCL then iterates through two steps: *Motion and Sensor update* and *Resampling*.

2.3.1 Motion and Sensor update

During *motion and sensor update* a hypothetical state is computed whenever the robot moves:

$$m = \text{current particle}$$

$$u_t = \text{robot actuation function}$$

$$x_t = \text{hypothetical state}$$

$$x_t^m = \text{motion\_update}(u_t, x_{t-1}^m)$$

In succession the particles new weight is calculated using the latest measurement:

$$z_t = \text{sensor measurements}$$

$$w_t^m = \text{sensor\_update}(z_t, x_t^m)$$

The new belief distribution is the sum of the prior belief distribution and the newly calculated particles:

$$X_t = X_t + < x_t^m, w_t^m >$$

2.3.2 Resampling

In the *resampling* stage of the algorithm particles with low probabilities are removed from the distribution. The remaining particles survive and can be drawn in a simulation tool. Using *Bayer Filtering* a probability density over the state space is estimated.

$$\text{Belief}(X_t) = P(X_t|Z_{1..t})$$

The result is a particle distribution which represents the new belief of the robots position.

3 Model Configuration

Hint: The finished model can be viewed in the following repository, folder */urdf*: https://github.com/yulivee/RoboNDWhereAmI

Most of the parameters were acquired experimentally by making use of `rqt_reconfigure`, which is able to edit the values of the ROS parameter during a running ROS session. This made it possibly to quickly see the results of changes in the robots behaviour. Restarting the whole setup was only necessary when the position of the robot had to be reset or changes had been made to the robot urdf.

After aquiring experience in parameter tuning with the provided robot, a new model is designed from scratch. Inspired by popular vacuum cleaning robots the base was chose to be of cylindrical shape. The size was kept close to the provided robot as the map featured narrow hallways a bigger robot could easily get stuck in. For a wider view of the environment the laser sensor was moved to the back of the model. As the camera had no impact on AMCL and just posed as a visual aid in debugging the robots actions it was left to the front by changed to a cylinder for optical reasons. The wheels were made smaller and places beneath the robot, the caster wheels were kept. Initially the robot featured only 3 wheels, one front caster and two wheels in the last third of the robots base. This setup proved to be impractical: the robot was unable to rotate on the spot which made localization when driving curves very uncertain. The wheels were moved back to the middle and the problem immediately vanished. The finished robot can be seen in 2

3 shows a representation of the model in Rviz.
The teal blue layer around the map obstacles is the *inflation radius* of the global costmap. It is tuned to 0.2m to prevent the global path to be planned to narrow along the map obstacles. This is especially important when navigating around the poles of the map.
The pink layer on the map is the *inflation radius* of the local costmap. By tuning this parameter to 0.15 the robot got stuck on walls less often.
The red dots are the collisions with the laser beam against the maps walls. It nicely overlays with the map localized by amcl, proving that the estimate of the position is a good one.
Beneath the robot the particle cloud of amcl can be seen as a cloud of teal blue arrows. The number of particles was reduced to 1000 to optimize the calculation time in the provided VM.
The blue path in front of the robot are the vanishing remains of its global plan before it rotated into the position visible on the map.

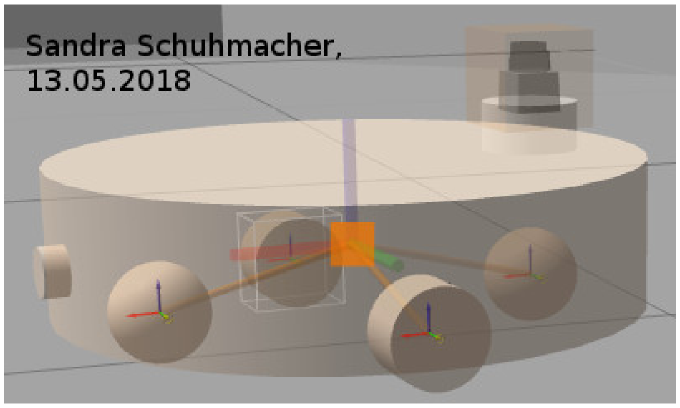


Fig. 2. Robot Definition from Gazebo

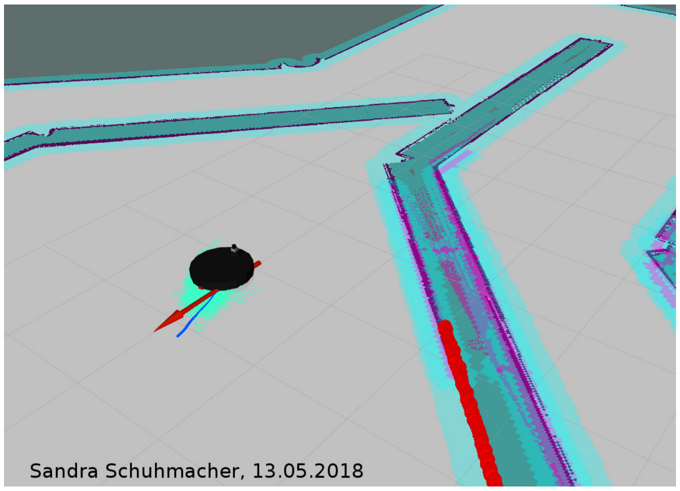


Fig. 3. Robot in Rviz Simulator

3.1 Robot base inertia and friction definitions

It is important to properly define the inertia and friction values of the model. In the provided robot, the values deviated significantly which leads to difficult behaviour. For example with the wrong friction values, the robot drives very slowly and often shows a behaviour where it osscilates between left and right movements. Wrong inertia values lead to overshooting due to wrong internal calculations with acceleration and velocity. Overshooting often leads to a deviation from the global path and negativly impacts the loclization. So the inertia values were recalculated for both models. As the models feature different base shapes, different inertia models had to be chosen [1].

1. Source for inertia calculations https://en.wikipedia.org/wiki/List_of_moments_of_inertia

Chassis inertia for provided robot (Solid cuboid of height h, width w, and depth d, and mass m)

$$m = 15 \quad (1)$$
$$w = 0.2 \quad (2)$$
$$h = 0.1 \quad (3)$$
$$d = 0.4 \quad (4)$$
$$(5)$$
$$I_h = \frac{m}{12} \cdot (w^2 + d^2) \quad (6)$$
$$I_w = \frac{m}{12} \cdot (d^2 + h^2) \quad (7)$$
$$I_d = \frac{m}{12} \cdot (w^2 + h^2) \quad (8)$$
$$(9)$$
$$I_h = \frac{15}{12} \cdot (0.04 + 0.16) = 0.25 \quad (10)$$
$$I_w = \frac{15}{12} \cdot (0.16 + 0.01) = 0.2125 \quad (11)$$
$$I_d = \frac{15}{12} \cdot (0.04 + 0.01) = 0.0625 \quad (12)$$
$$(13)$$

Chassis inertia for self constructed robot (Solid cylinder of radius r, height h and mass m)

$$I_z = \frac{1}{2} \cdot m \cdot r^2 \quad (14)$$
$$I_x = I_y = \frac{1}{2} \cdot m \cdot (3 \cdot r^2 + h^2) \quad (15)$$
$$(16)$$
$$m = 15 \quad (17)$$
$$h = 0.15 \quad (18)$$
$$r = 0.3 \quad (19)$$
$$(20)$$
$$I_z = \frac{15 \cdot 0.3^2}{2} = 0.675 \quad (21)$$
$$I_x = I_y = \frac{15}{12} \cdot (3 \cdot 0.3^2 + 0.15^2) = 0.365625 \quad (22)$$
$$(23)$$

The inertia values for the wheels of both robots are calculated in a similar fashion as they are also cylinders. The caster wheels of both robots are calculated with the spheroid model. Shown below is the ineratia calculation for the self constructed robot, the calculation of the casters of the provided robot was done in a similar fashion.

Caster wheel inertia for self constructed robot (Solid sphere (ball) of radius r and mass m)

$$I_z = I_x = I_y = \frac{2}{5} \cdot m \cdot r^2 \quad (24)$$
$$m = 2 \quad (25)$$
$$r = 0.05 \quad (26)$$
$$I_z = I_x = I_y = \frac{2}{5} \cdot 2 \cdot 0.05^2 = 0.002 \quad (27)$$

3.2 move base - ROS Navigation Stack

The move base navigation stack features two planners - a global planner which is doing a long term path plan to a given goal on the map and takes the parameters of the global costmap into account for its planning behaviour and obstacle avoidance. It also operates on global localization. The local plan calculates a path along the global path taking into account the robots current location provided by odometry and using its own local costmap for collision avoidance. Both planners also share a recovery behaviour once the robot gets stuck. The costmaps are calculated by using information of a laser scanner or point cloud sensor and the map data.

In the following sections only parameters that deviate from the default are listed.

3.2.1 Base Local Planner

The base local planner configuration defines the robot behaviour while locally trying to follow the global plan in the plane. The parameter `pdist_scale` controls how close the local planner follows the global plan and how much deviation is tolerated. The `gdist_scale` controls speed and how much effort is put into reaching the goal. Experiments showed that a high value around 2.2 is needed to let the robot reach the exact final navigation goal or else it would just stop navigating when nearby. The `meter_scoring` was activated to calculate the two parameters in meters. Along with those, the parameters `xy_goal_tolerance` and `yaw_goal_tolerance` were tuned to allow some tolerance in navigation in x,y and yaw-rotation. Setting those to strict resulted in a robot constantly driving past its goal, turning and trying again.

The parameter `holonomic_robot` was set to false as none of the robots featured omni-wheels and both had movement restrictions. The `simtime` is the timespan to forward simulate trajectories - it influences the size of the local plan, as more forward simulations means more planning ahead. A higher value also resulted in steadier movements as the robot made fewer corrections while driving. The update frequency of the controller `controller_frequency` was set to a low value to reduce the amount of *Controller loop was missed* warning messages in the simulation. The velocity values were tuned to a value were the robot would travel at a sufficient speed but neither overshoot when driving curves nor crash into obstacles because it couldn't reduce speed fast enough. Setting /texttescape_vel to a negative value allows the robot to drive reverse when getting stuck on an obstacle.

```
TrajectoryPlannerROS:
  pdist_scale: 2.2
  gdist_scale: 2.2
  sim_time: 3.0
  yaw_goal_tolerance: 0.15
  xy_goal_tolerance: 0.30
  holonomic_robot: false
  controller_frequency: 5.0
  meter_scoring: true
  min_vel_x: 0.1
  max_vel_x: 2.0
  max_vel_theta: 1.0
  min_in_place_vel_theta: 0.4
  acc_lim_theta: 2.2
  acc_lim_x: 1.0
  acc_lim_y: 1.0
  global_frame_id: odom
  escape_vel: -0.6
```

3.2.2 Shared Costmap Parameters

The shared costmap parameters are taken into account by both the local and the global planner. The two most important parameters in this section are `robot_radius`, which defines the robots base radius for the obstacle calculation. Enlarging this value will lead to the robot leaving more space between itself and an obstacle (or its inflation radius). It was set to the same size as the bases actual collision values. For the provided robot `robot_footprint` was used instead with an array of edge-points instead. The `transform_tolerance` parameter sets the maximum amount of latency allowed between two transformations from the robots global frame (the map) to the robots base frame (the odom values). If the transformations do not occur within this timeframe, the robot is stopped. A typical sign of a too small transform_tolerance is a robot that stops a lot intermittingly and starting to drive again.

```
map_type: costmap
robot_base_frame: robot_footprint
transform_tolerance: 0.7
robot_radius: 0.3
observation_sources: laser_scan_sensor
laser_scan_sensor:
{sensor_frame: hokuyo, ...}
```

3.2.3 Global Costmap Parameters

The global costmap is calculated based on the input of the laser sensor and takes the map as a global frame. The most important parameters during costmap tuning proved to be the `inflation_radius`. It influences how close the robot may navigate to a wall before it gets a cost penalty. In practice this can be seen in how close to the walls the blue line of the global plan is drawn in Rviz. Setting this parameter to a proper value made the robot plan a route far enough from the poles at the edges to avoid getting into collision problems when calculating the local plan. A larger global costmap allows for more planning ahead, hence the bigger values for `width` and `height`.

```
global_costmap:
   global_frame: map
   publish_frequency: 5.0
   width: 40.0
   height: 40.0
   inflation_radius: 0.2
```

3.2.4 Local Costmap Parameters

The global costmap is calculated based on the input of the odometry and takes the odom as a global frame. The `rolling_window` is set to true because the map changes with the updating of the position provided by the amcl. The `inflation_radius` is set to a lower value to allow the robot to drive closer to walls when navigating.

```
local_costmap:
   global_frame: odom
   publish_frequency: 2.0
   width: 20.0
   height: 20.0
   rolling_window: true
   inflation_radius: 0.15
```

3.3 AMCL Parameters

The amcl parameters were outsourced to `config/amcl_params.yaml` for better readability. To reduce calculation overhead and get a less bigger cloud of particles, `max_particles` was reduced to 1000. Those were enough parameters for a good localization and had the nice benefit that some of the outliers with low weights vanished from the distribution. The most important parameters in this section proved to be the `update_min` parameters which define the minimum required movement and rotation before a filter update in the amcl. If any laser measurement arrives in between it is discarded. This influences the speed of the amcl position updates.

```
base_frame_id: robot_footprint
global_frame_id: map
odom_frame_id: odom
odom_model_type: diff-corrected
max_particles: 1000
min_particles: 100
laser_model_type: likelihood_field
laser_sigma_hit: 0.2
laser_z_hit: 0.99
laser_z_rand: 0.01
update_min_a: 0.131
update_min_d: 0.05
```

4 RESULTS

Both robots were able to reach their goal position. The cylindrical robot proved to be able to go faster without overshooting on the local plan. It was key to get a good initial pose estimate to get the robots to start navigation. This is done by using the *2D Pose Estimate* feature of Rviz and clicking into the center of the robots base. The initially wide distributed amcl particle cloud became more dense. Without this, the robots would sometimes even start navigation by driving into the nearest wall.

5 DISCUSSION

Both robots were able to navigate and localize themselfes within a reasonable time frame and accuracy. The self-constructed robot was able to drive faster while navigating as accurately as the provided robot. The smaller wheels helped with more precise rotations. Also, as the hokuyo sensor was moved to the back of the robots base, it had a broader view of its environment which made the localization more precise. The provided model tended to recognise its own wheels as an obstacle by the hokuyo sensor. The sensor had to be moved above the wheels to accomodate for this problem.

The round shape of the self-constructed robot was able to get itself easier unstuck, the provided robot could get stuck at its own edges of its box shaped shape.

6 Conclusion / Future work

To increase accuracy an additional point cloud sensor could be mounted onto the robot to have a second source of information for the global costmap input. Also some further fine tuning could be done on the acceleration and velocity-values to figure out the fastest way to drive without impacting the navigation.

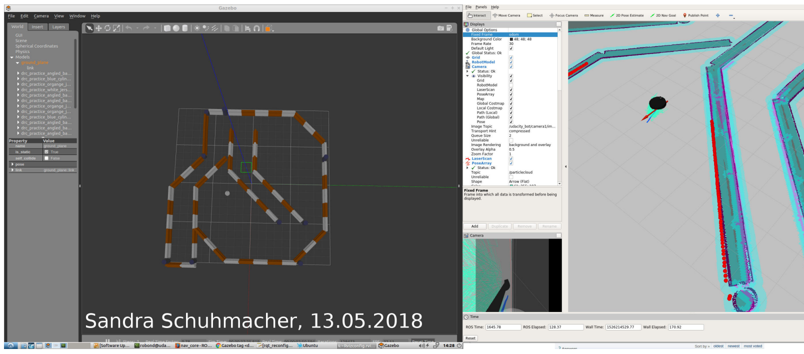
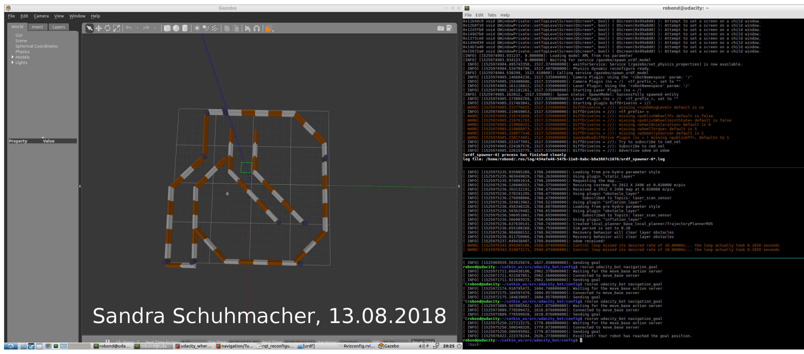


Fig. 4. Own Robot at Goal Position



```
0x11b4dc0 void QWindowPrivate::setTopLevelScreen(QScreen*, bool) ( QScreen(0x99a8d0) ): Attempt to set a screen on a child window.
0x12b0f10 void QWindowPrivate::setTopLevelScreen(QScreen*, bool) ( QScreen(0x99a8d0) ): Attempt to set a screen on a child window.
0x12d3fb0 void QWindowPrivate::setTopLevelScreen(QScreen*, bool) ( QScreen(0x99a8d0) ): Attempt to set a screen on a child window.
0x149d7b0 void QWindowPrivate::setTopLevelScreen(QScreen*, bool) ( QScreen(0x99a8d0) ): Attempt to set a screen on a child window.
0x1373ce0 void QWindowPrivate::setTopLevelScreen(QScreen*, bool) ( QScreen(0x99a8d0) ): Attempt to set a screen on a child window.
0x149dd30 void QWindowPrivate::setTopLevelScreen(QScreen*, bool) ( QScreen(0x99a8d0) ): Attempt to set a screen on a child window.
0x14b7a40 void QWindowPrivate::setTopLevelScreen(QScreen*, bool) ( QScreen(0x99a8d0) ): Attempt to set a screen on a child window.
0x15672a0 void QWindowPrivate::setTopLevelScreen(QScreen*, bool) ( QScreen(0x99a8d0) ): Attempt to set a screen on a child window.
[INFO] [1525974903.931237, 0.000000]: Loading model XML from ros parameter
[INFO] [1525974903.934123, 0.000000]: Waiting for service /gazebo/spawn_urdf_model
[ INFO] [1525974904.495743350, 1517.374000000]: waitForService: Service [/gazebo/set_physics_properties] is now available.
[ INFO] [1525974904.534784790, 1517.407000000]: Physics dynamic reconfigure ready.
[INFO] [1525974904.538299, 1517.410000]: Calling service /gazebo/spawn_urdf_model
[ INFO] [1525974905.146604238, 1517.535000000]: Camera Plugin: Using the 'robotNamespace' param: '/'
[ INFO] [1525974905.155480686, 1517.535000000]: Camera Plugin (ns = /)  <tf_prefix_>, set to ""
[ INFO] [1525974905.161116822, 1517.535000000]: Laser Plugin: Using the 'robotNamespace' param: '/'
[ INFO] [1525974905.161181261, 1517.535000000]: Starting Laser Plugin (ns = /)
[INFO] [1525974905.162812, 1517.535000]: Spawn status: SpawnModel: Successfully spawned entity
[ INFO] [1525974905.173004769, 1517.535000000]: Laser Plugin (ns = /)  <tf_prefix_>, set to ""
[ INFO] [1525974905.217463841, 1517.535000000]: Starting plugin DiffDrive(ns = //)
[ WARN] [1525974905.217776832, 1517.535000000]: DiffDrive(ns = //): missing <rosDebugLevel> default is na
[ INFO] [1525974905.219659053, 1517.535000000]: DiffDrive(ns = //): <tf_prefix> =
[ WARN] [1525974905.219743608, 1517.535000000]: DiffDrive(ns = //): missing <publishWheelTF> default is false
[ WARN] [1525974905.219761763, 1517.535000000]: DiffDrive(ns = //): missing <publishWheelJointState> default is false
[ WARN] [1525974905.219868431, 1517.535000000]: DiffDrive(ns = //): missing <wheelAcceleration> default is 0
[ WARN] [1525974905.219888973, 1517.535000000]: DiffDrive(ns = //): missing <wheelTorque> default is 5
[ WARN] [1525974905.220077548, 1517.535000000]: DiffDrive(ns = //): missing <odometrySource> default is 1
[ WARN] [1525974905.220174881, 1517.535000000]: GazeboRosDiffDrive Plugin (ns = ) missing <publishTf>, defaults to 1
[ INFO] [1525974905.221477691, 1517.535000000]: DiffDrive(ns = //): Try to subscribe to cmd_vel
[ INFO] [1525974905.224387576, 1517.535000000]: DiffDrive(ns = //): Subscribe to cmd_vel
[ INFO] [1525974905.226153775, 1517.535000000]: DiffDrive(ns = //): Advertise odom on odom
[urdf_spawner-6] process has finished cleanly
log file: /home/robond/.ros/log/434afe46-547b-11e8-8abc-b8a3867c1676/urdf_spawner-6*.log

[ INFO] [1525975235.935085289, 1768.249000000]: Loading from pre-hydro parameter style
[ INFO] [1525975235.965049829, 1768.263000000]: Using plugin "static_layer"
[ INFO] [1525975235.974891014, 1768.269000000]: Requesting the map...
[ INFO] [1525975236.126606553, 1768.375000000]: Resizing costmap to 2912 X 2496 at 0.020000 m/pix
[ INFO] [1525975236.263132191, 1768.475000000]: Received a 2912 X 2496 map at 0.020000 m/pix
[ INFO] [1525975236.270191295, 1768.477000000]: Using plugin "obstacle_layer"
[ INFO] [1525975236.276098066, 1768.479000000]:     Subscribed to Topics: laser_scan_sensor
[ INFO] [1525975236.324813962, 1768.521000000]: Using plugin "inflation_layer"
[ INFO] [1525975236.456348326, 1768.607000000]: Loading from pre-hydro parameter style
[ INFO] [1525975236.503629482, 1768.653000000]: Using plugin "obstacle_layer"
[ INFO] [1525975236.506951681, 1768.655000000]:     Subscribed to Topics: laser_scan_sensor
[ INFO] [1525975236.566083929, 1768.694000000]: Using plugin "inflation_layer"
[ INFO] [1525975236.637630141, 1768.743000000]: Created local_planner base_local_planner/TrajectoryPlannerROS
[ INFO] [1525975236.655108260, 1768.755000000]: Sim period is set to 0.10
[ INFO] [1525975236.904888152, 1768.942000000]: Recovery behavior will clear layer obstacles
[ INFO] [1525975236.911750966, 1768.949000000]: Recovery behavior will clear layer obstacles
[ INFO] [1525975237.049438407, 1769.044000000]: odom received!
[ WARN] [1525976342.894585106, 2568.674000000]: Control loop missed its desired rate of 10.0000Hz... the loop actually took 0.1010 seconds
[ WARN] [1525976343.919873171, 2569.475000000]: Control loop missed its desired rate of 10.0000Hz... the loop actually took 0.1020 seconds

[ INFO] [1525969939.563525674, 1627.458000000]: Sending goal
robond@udacity:~/catkin_ws/src/udacity_bot/config$ rosrun udacity_bot navigation_goal
[ INFO] [1525971711.666430186, 2962.378000000]: Waiting for the move_base action server
[ INFO] [1525971711.921587851, 2962.560000000]: Connected to move_base server
[ INFO] [1525971711.921690272, 2962.560000000]: Sending goal
^Crobond@udacity:~/catkin_ws/src/udacity_bot/config$ rosrun udacity_bot navigation_goal
[ INFO] [1525972174.910795472, 1604.748000000]: Waiting for the move_base action server
[ INFO] [1525972175.184597479, 1604.957000000]: Connected to move_base server
[ INFO] [1525972175.184639697, 1604.957000000]: Sending goal
^Crobond@udacity:~/catkin_ws/src/udacity_bot/config$ rosrun udacity_bot navigation_goal
[ INFO] [1525973889.507885562, 1617.875000000]: Waiting for the move_base action server
[ INFO] [1525973889.778509472, 1618.076000000]: Connected to move_base server
[ INFO] [1525973889.778599928, 1618.076000000]: Sending goal
^Crobond@udacity:~/catkin_ws/src/udacity_bot/config$ rosrun udacity_bot navigation_goal
[ INFO] [1525975250.227113175, 1778.866000000]: Waiting for the move_base action server
[ INFO] [1525975250.500548326, 1779.073000000]: Connected to move_base server
[ INFO] [1525975250.500595992, 1779.073000000]: Sending goal
[ INFO] [1525976422.222378244, 2626.273000000]: Excellent! Your robot has reached the goal position.
robond@udacity:~/catkin_ws/src/udacity_bot/config$
```

Fig. 5. Provided Robot at Goal Position