# Report: Where Am I Project

Mar 5, 2018

## Abstract

This work explores the process of building robot model using Unified Robot Description Format (URDF) with Gazebo sensors and actuators plugins that can be used for a further simulation. Standard ROS Navigation stack is used to navigate a robot towards the specified goal using the static map. For localization, the corner stone of the robot navigation, Adaptive Monte Carlo Localization algorithm was used and compared with the Kalman-Filter based algorithms. Tuning parameters strategy is described in detail and future possibilities discussed.

## Introduction

Robotics software development tightly bounded with the hardware and system levels. Due to the lack of (hard to achieve, impractical) standard hardware form-factor roboticists should first devise the hardware representation of it's model with the desired parameters in form, actuators, sensors, load capacity and other requirements. These parameters influence dynamics configuration of a robot which are usually necessary to know for a localization algorithms. For example wheel diameters and it's distance need to be feed to the odometry algorithm and differential driver. Cameras intrinsic and extrinsic parameters need to be known before we can use it for a perception layer.

First important characteristic of a robot is to know it's location in a surrounding space so it can reason about neighboring features and build plans for the movement goals. Two common types of localization are recognizes: local and global.

Local localization is determining robot pose and orientation based on odometry and perception sensors data before any matching to the map.

Global localization, on the other hand, combines output from local localization and matches it to the known map so to place the robot in exact position and orientation on the map. In other word it is describing the translation and rotation from `odom` frame to the `map` frame.

In this work we are building the basic URDF model of a robot, which represents the full links geometry descriptions along with inertial, visual and connection parameters.

Then we are showing how to test it using `rostopic pub` and `teleop` commands and verify the driver performance and processing of `cmd_vel` commands.

Later we are adding Adaptive Monte Carlo Localization `amcl` algorithm and test it in ‘RViz’. After that we are adding full capabilities from ROS Navigation stack and making robot plan and navigate on a provided static map (ROS `move_base` )

# Background

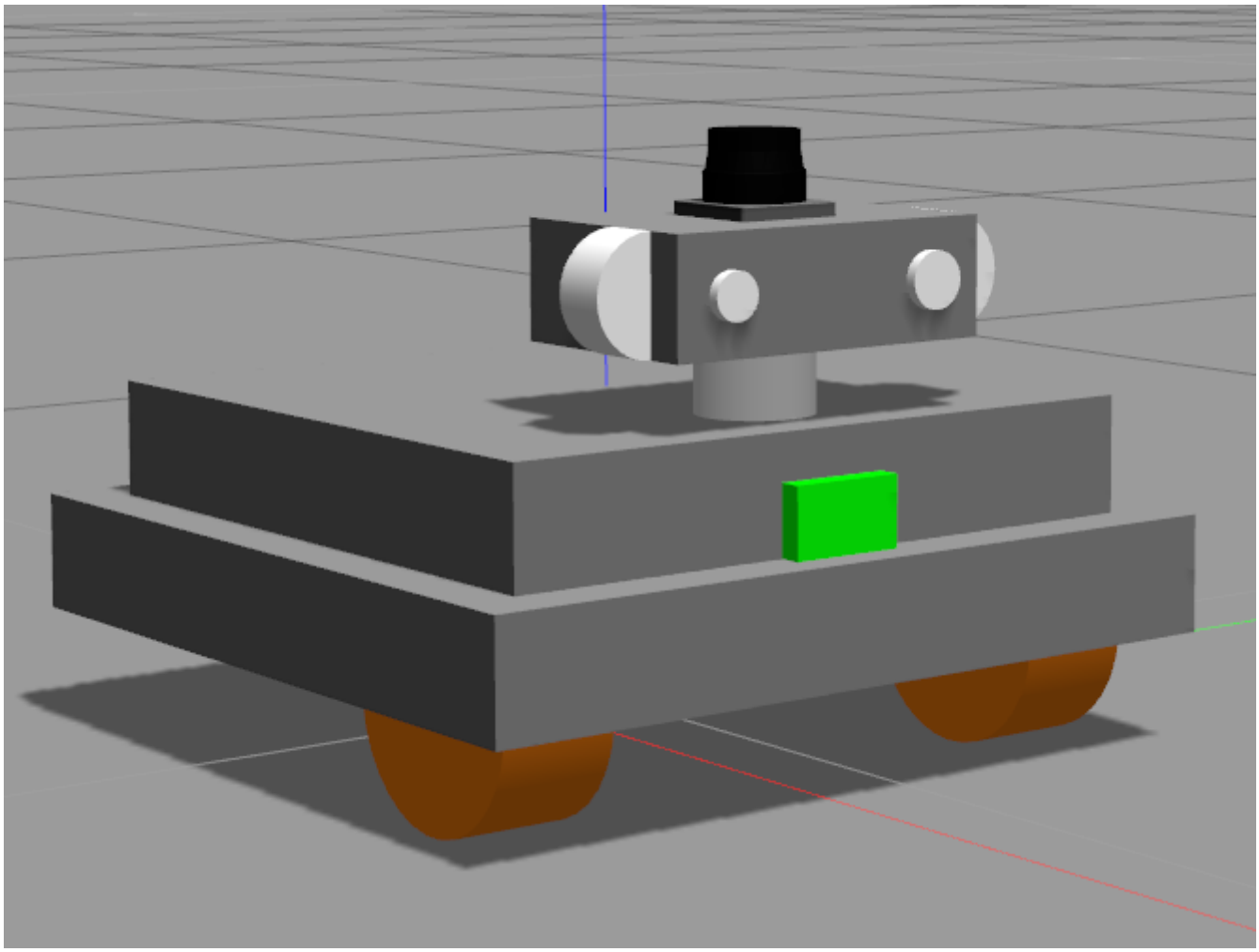
Kalman filters considered a sufficiently accurate algorithm that works great for small incremental errors that arise due to the sensors errors and motion models discrepancies (noise representation, wheel slippage etc.) like odometry estimation and local robot pose. However Gaussian assumption employed by Kalman filters is restrictive for global localization problem.

Monte Carlo Localization algorithms based on sampled particles and thus can represent any distribution function, that is not limited by a Gaussian model and is not linear. MCL is also much simpler in calculation and implementation compare to Kalman filters algorithms. MCL is also a good choice when one can want to control memory and resolution control of a filter which can necessary for the small embedded computers with limited resources.

# Model Configuration

## Custom URDF Robot

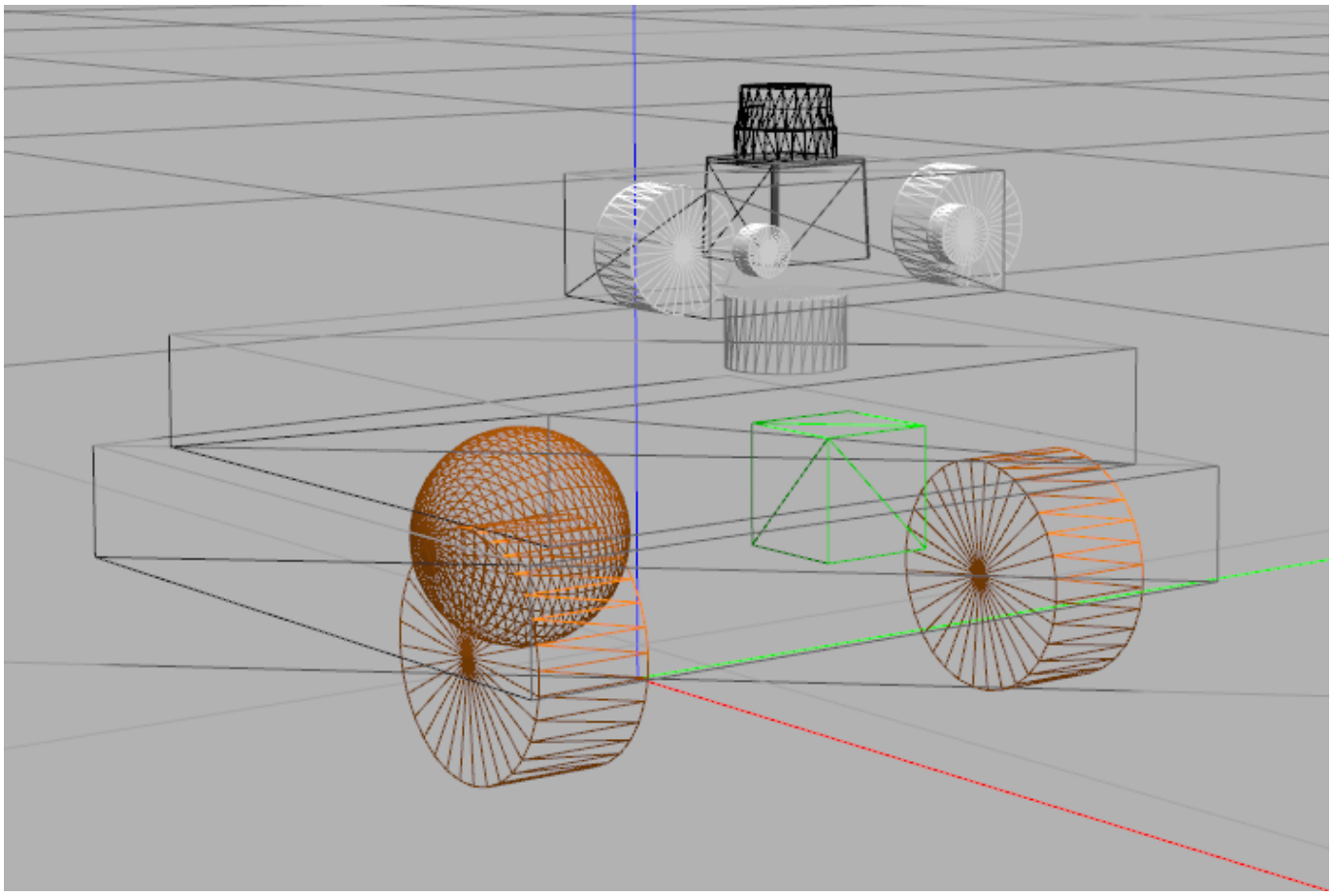
Custom robot was built based on the square platform with two wheels and one caster. Thus, control base is a standard differential drive.



Standard package of sensors was chosen:

- Laser Scanner (Hokuyou), placed on the had in order to have a free space in 180 degrees in front of a robot.
- Camera, placed in front to see what's ahead (however, in this work camera sensors was not used for navigation purposes)

Wireframe look of the model provided below:



## AMCL Configuration

Adaptive Monte Carlo Localization algorithm uses variable number of particles depending on certainty which helps free computational resources. For the given task as small number of particles as `100` works too but it's can turn bad in an uncertain events and failure when we will not have enough particles distributed over the space.

```
<param name="min_particles" value="50" />
<param name="max_particles" value="2000" />
```

The longevity of transform was set up to `0.3` seconds which was derived experimentally from the time of processing on test machine.

```
<param name="transform_tolerance" value="0.3" />
```

Our robot starts at the center of the map so it's pose was set up `[0, 0]`

```
<param name="initial_pose_x" value="0.0" />
<param name="initial_pose_y" value="0.0" />
<param name="initial_pose_a" value="0.0" />
```

With the `odom_model_type=diff-corrected` standard values for noise in odometry rotation and translation estimates is not working well and should be decreased.

```
<param name="odom_alpha1" value="0.002" />
<param name="odom_alpha2" value="0.002" />
<param name="odom_alpha3" value="0.002" />
<param name="odom_alpha4" value="0.002" />
```

# Move Base Configuration

Move base configuration includes the common parameters and local/global costmap parameters.

For common parameters was selected values that works well for two robots simultaneously. For example, `footprint` of custom robot ( `my` ) is a bit smaller than standard udacity but a bit higher values works great for both.

`transform_tolerance` was set up in accordance to the AMCL configuration.

`inflation_radius` was lowered from `0.55` default value to the closer of the robot shape.

```
transform_tolerance: 0.3

# (Udacity Robot)
footprint: [[-0.22, -0.20], [-0.22, 0.20], [0.22, 0.20], [0.22, -0.20]]

# (My Custom Robot)
# footprint: [[-0.18, -0.18], [-0.18, 0.18], [0.18, 0.18], [0.18, -0.18]]

inflation_radius: 0.4
```

For local costmap the size of the map was lowered to `5` meters because it solved problem when robot moves in opposite direction of the global path in an attempt to reach goal through the wall.

In `local_costmap_params.yaml`:

```
width: 5.0
height: 5.0
```

In order to be able to unstack we should set params like `min_vel_x`, `min_in_place_vel_theta` and `escape_vel` to overcome the friction and mass inertia.

Also, it was proven useful to setup goal tolerance values in order to stop robot rotation and movement around the goal and not continuously overshoot it.

In `base_local_planner_params.yaml`:

```
min_vel_x: 0.15
min_in_place_vel_theta: 0.3
escape_vel: -0.25

# Goal Tolerance Parameters
xy_goal_tolerance: 0.15
yaw_goal_tolerance: 0.05
```

# Results

To easily test navigation the launch scripts was combined in one `udacity_test.launch` with an optional parameter `bot` so we can run the whole sequence using commands:

For standard bot:

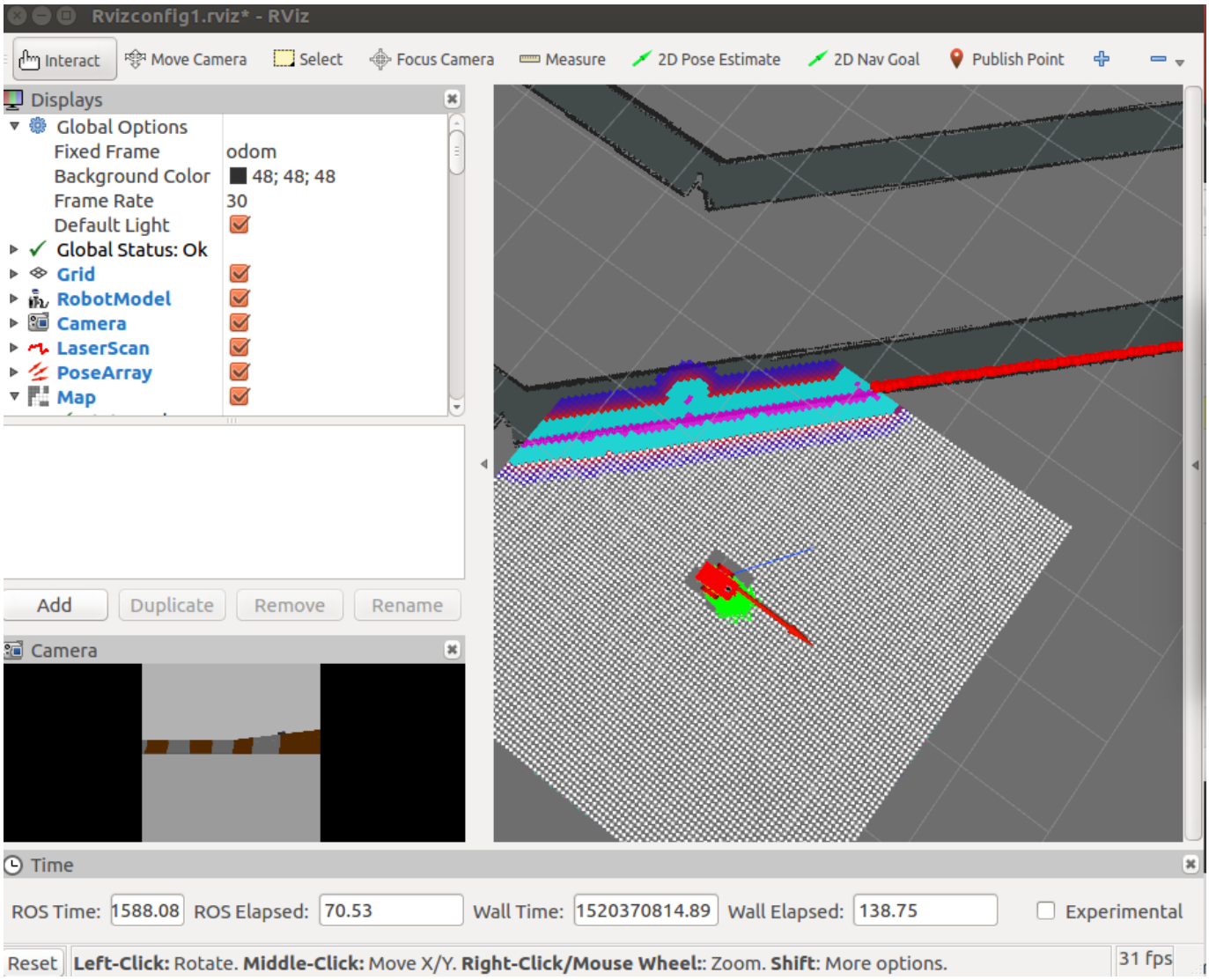
```
roslaunch udacity_bot udacity_test.launch
```

and for the custom bot:

```
roslaunch udacity_bot udacity_test.launch bot:=my
```

## Standard Classroom Robot Results

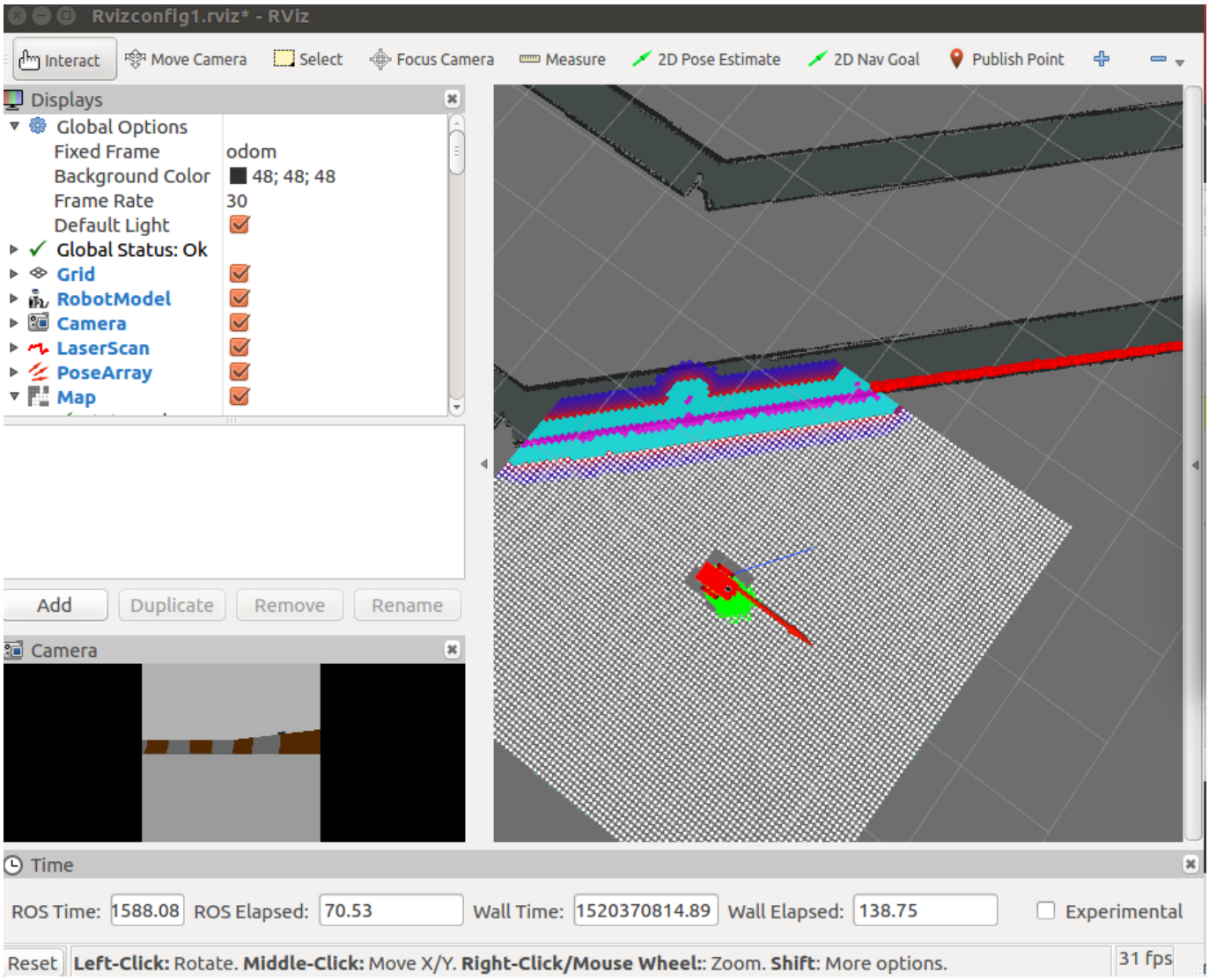
Below is an image with the final result that also linked to the video of the robot navigation:




Video Link: https://youtu.be/WiRRt13snP8

# Custom (My) Robot Results

Custom robot moves almost the same way as classroom robot with an occasional backing up to recover from the close up with the wall after initial rotation.




Video Link: https://youtu.be/qMKT9Q2fBQ0

# Discussion

For AMCL configuration `odom_alpha` params was crucial in order to achieve a normal estimation of the robot pose, without them the deviation was too high.

As for the move base param configuration it was much trickier because lot of parameters is scattered among different layers. And if one started to quick a lot of parameters simultaneously the resulting behavior was not converging at all. But starting from scratch by commenting out all params (even default values) was proven helpful. Eventually navigation started working with much lower number of configured parameters.

As for the learning, capping up `max_vel_x` to some value (like `0.7-1.0`) was leading to the halting in local path planner so robot were not moving smooth and made a full stop right in the middle of the path.

Lot of useful params were reverted back after a long hours of experimentation because defaults showed better results than any tweaks. (`sim_time`, `vx_samples`, `pdist_scale`, `gdist_scale`, `occdist_scale` and others).

The problem of a kidnapped robot is not solved by the plain AMCL algorithm because once it converges around estimated robot pose there almost no particles left in a distant areas of the map so if robot were kidnapped in a such area without particles it would not be able to recover. One of the possible solution is to built a detector of the failure like comparing subsequent sensor measurements and if the deviation is a complete anomaly we can reset AMCL state to the uniform distribution and re-estimate the robot pose from this. Different methods of kidnapping event were proposed in the literature, like weight particle distribution or entropy of information that could be extracted from the weights.

Monte Carlo Localization can be used for the household robot with the known map at home or in the office space. However, moving people and furniture would pose troubles in the localization due to the changing map, which can be addressed by building dynamics maps of the environment.

# Future Work

For the current environment and robot configuration one of the possible improvement could be to replace a 180 degree laser with a 360 laser scanner. It will cover more space and will provide more features for the AMCL algorithms.

Other possible improvement that was planned for a future work on custom model is to add a 3D structural depth sensor as eyes which can be useful for another level of feature extraction and/or fine grained object detections (like toys, fruits, missed wedding ring, etc).

As for the ears, we can use microphones and than process it as a spatial sound by extracting audio features that we can later feed to the AMCL or SLAM algorithms.

---


Capsules Bot

Pavlo Bashmakov

bexcite

bashmakov

Exploring autonomous systems, robotics and mapping world around us.