Project "Where Am I"

Dmitry Gavrilenko

Abstract – This paper describes the process of a two-wheeled robot model design in Robot Operating System (ROS). The model physics is simulated in Gazebo environment. Its position is localized in a predefined map with a particle filter, implemented by amcl ROS package. A differential_drive_controller, following local paths generated by base_local_planner/TrajectoryPlannerROS package, controls its motion. Local paths are evaluated to be as close as possible to global paths of navfn/NavfnROS package.

Index – ROS, particle filter, navigation, amcl, path planning, localization

1 Introduction

Robotic Operating System (ROS) is a framework intended to facilitate robot design, development and debugging. It consists of a physical simulator, hardware drivers, communication protocols and a set of standard algorithms for sensor fusion, computer vision, inverse kinematics, localization and path planning.

The robotic software in ROS is split into multiple processes, called nodes. Nodes exchange messages by subscribing to or publishing topics. Each topic is a communication channel with a predefined message format.

In order to create a mobile robot model in a ROS physical simulator, the developer needs to set up a number of configuration files, defining the model geometry, its physical properties and parameters of nodes, responsible for localization, control and path planning.

2 Background

Localization is the process of determining the position and orientation of a robot with respect to the map. Since the real world and robot motion are noisy, the localization process is inherently probabilistic.

There are two popular probabilistic algorithms for localization: Kalman Filter and Particle Filter. Both are based on Bayesian Inference theory and Markov Assumption (see also [1], [2] and [3]).

Bayesian Inference approach treats probability as the amount of uncertainty about the state of the world. Its foundational formula is:

$$P(s|o) = \frac{P(o|s) * P(s)}{P(o)} \quad (1)$$

Where $P(o)$ is the probability of some observation (e.g. sensor values), $P(s)$ is the probability of a state (e.g. position of a robot), $P(s|o)$ is the probability of a state given sensor values, and $P(o|s)$ is the probability of sensor values given a state.

The formula (1) explains how to predict the probability of a state, given sensor measurements, the previous state and some initial statistics about what states produce what measurements. In case of a robot localization, the probabilities are usually continuous values. The formula (1) turns into multiplications of multi-dimensional integrals, which can be approximated by Gaussian distributions (Kalman Filter) or set of randomly generated particles (Particle Filter).

Markov Assumption assumes that the probability of a state s_i depends only on the previous state s_{i-1} and the current observations o_i. All the previous history of states and observations is thus ignored. This removes a lot of redundant calculations and enables development of efficient algorithms that work well in practice.

Kalman Filter is one of such algorithms, with the following expectations from states and observations:

- States and observations must be continuous probabilistic values
- Each new state linearly depends on the previous state and current observations; or these dependencies can be linearly approximated
- State and observation distributions are Gaussian or can be approximated by Gaussians

Gaussian distributions are *unimodal*. Unimodality assumes there is only one most probable value, which is located in the center of the distribution.

Unimodal property of Gaussian Filter limits the applicability of the classical implementation of the Kalman Filter: it can be used only for object tracking (inferring position of a robot from the previously known position and observations), but not for localization (inferring position of a robot only from observations, where multiple potential positions, corresponding to the same observations, exist).

Kalman Filter has $O(n^3 + m^2)$ time efficiency, where n is the state size, and m is the observation size.

Particle Filter, in turn, has the following expectations:

- States and observations must be continuous values
- Each new state may non-linearly depend on the previous state and current observations
- States and observation distributions do not have to be Gaussians. They may also be multimodal. That is, they may have multiple local maximums

Multimodality implies Particle Filter may be applied to a robot localization task, in which the position of a robot within a map is initially unknown and initial observations correspond to multiple positions or orientations in different parts of the map.

With some extensions, Particle Filter may also be used to solve Kidnapping problem, when position of a robot was known, but then it was teleported to a new place so that its position can no longer be inferred from the previous one and should be determined only from observations from inception.

Properties, described so far, hint that Particle Filter outpaces Kalman Filter. However, Particle Filter has one significant disadvantage: its time and space complexity are of $O(p^n)$ where p is the number of particles along one state dimension, and n is the size of a state.

Since Particle Filter complexity exponentially grows with the state size, in practice, it usually cannot be applied to states, which dimensionality exceeds 4.

The robot, described in this paper, has 3 degrees of freedom: X and Y coordinate along the ground plane, and orientation yaw. Thus, it can be efficiently localized with Particle Filter, implemented in amcl ROS package.

3 Results

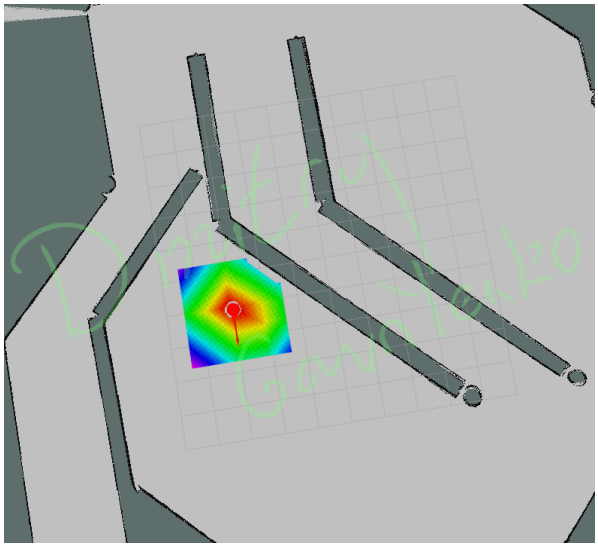

Fig. 1. my_bot reaching navigation_goal

Fig. 1 shows the robot, designed from inception within the scope of “Where Am I” project, which has reached the goal, sent by navigation_goal node.

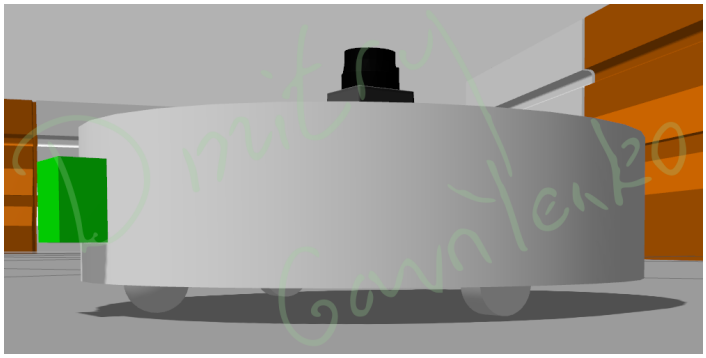

Fig. 2. my_bot side view

The robot model, shown in Fig. 2, reminds a vacuum cleaner robot with the LIDAR sensor installed in the top center to return 360 degree symmetric scan of the map. The camera is installed in the forward side of the cylindrical shape of the robot in order to avoid occlusion of LIDAR sensor beams.

The forward camera also increases the stability of the robot, shifting its center of mass forward so that only three points of contact with the ground are enough: one caster and two wheels.

The robot wheels are located to the left and to the right from the origin of the robot to enable pure rotational motion around its center if necessary.

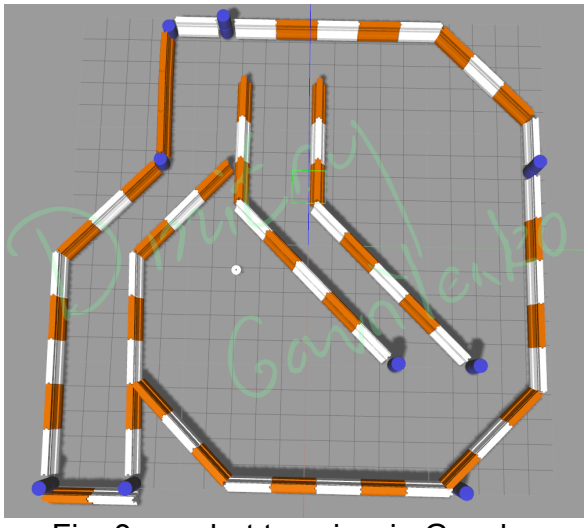

Fig. 3. my_bot top view in Gazebo

my_bot moves much faster than udacity_bot due to decreased weight and friction coefficients. Cylindrical shape simplifies local path planning task, because the orientation of the robot does not affect whether a map region is passable or not.

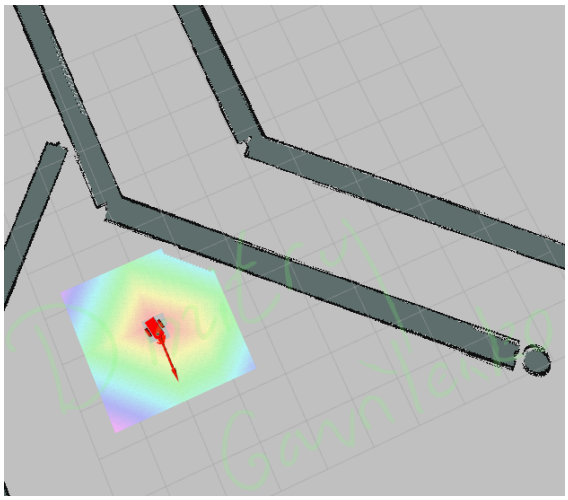

Fig. 4. udacity_bot reaching navigation_goal

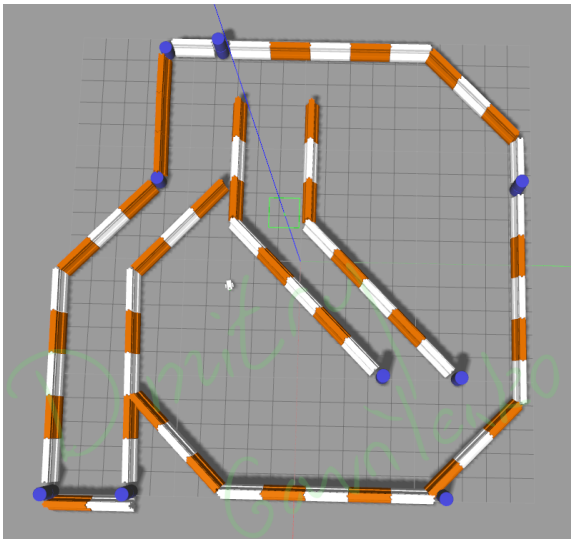

Fig. 5. udacity_bot top view in Gazebo

udacity_bot was also set up to work well in practice. Rviz screenshot, corresponding to the navigation_goal pose, is shown in Fig. 4. Gazebo screenshot, corresponding to the same moment, is in Fig. 5.

4 Model Configuration

Model configuration of my_bot consists of the following files:

./urdf/my_bot.xacro format is explained in [4] and ROS documentation. It defines the model visual and collision geometry in `<links>` as well as relationships between different parts of the robot in the form of `<joints>`. `<inertial>` values were calculated with a table from [12]. Initial tests with small `<inertial>` values,

corresponding to the mass and the shape of the object, produced numerical errors in Gazebo simulator. Small damping and friction coefficients for wheels, as well as zero friction values for the caster helped eliminate the numerical and stability issues.

./urdf/my_bot.gazebo defines additional properties of sensors and controllers installed in the robot. Two actuators are attached to left and right wheels correspondingly to control the robot motion. `<wheelSeparation>` defines the distance between the left and right wheels, and `<wheelDiameter>` defines the diameter of the wheels. The values are borrowed from `my_bot.xacro`. To prevent numerical stability issues with Gazebo, and to allow fast motion, `mu1` and `mu2` friction coefficients are set zero for the caster link, which is rigidly fixed to the chassis: ROS does not have a specific joint type for the caster, therefore it is simulated with the fixed joint and zero friction.

./config/my_base_local_planner_params.yaml defines the properties of the local path planner `TrajectoryPlannerROS`. `controller_frequency` has been decreased to 10 times per second as a good balance between performance and smoothness of the controller. `my_bot` cannot move sideways, therefore, it is non-holonomic robot (`holonomic_robot` is false). Simulation time is set to one second to enable smooth local trajectories (`sim_time` is 1.0). The local plan should be close to the global plan to move inside the labyrinth, also considering distance to goal and some obstacle avoidance (there are no obstacles in the testing scenario, though). Therefore, `pdist_scale` > `gdist_scale` > `occdist_scale`. All the distances are expressed in meters rather than cells (`meter_scoring` is true) to have some independency from the cost map resolution. To see the local cost map cells in Rviz, `publish_cost_grid_pc` is set true.

./config/my_costmap_common_params.yaml defines common parameters for local and global cost map (see also [8]). Cylindrical shape, with `robot_radius` equal 0.225, bounds both the robot chassis and its front camera. `inflation_radius` is set slightly higher than `robot_radius`, to 0.35, so as to keep the robot away from obstacles and avoid getting stuck in the corners of the labyrinth. `obstacle_range` and `raytrace_range` are set comparable to the size of the labyrinth. `transform_tolerance` is set to a small positive value to allow some latency of transform data update messages.

./config/my_global_costmap_params.yaml defines properties of the costmap for the global path planner. Here, `update_frequency` and `publish_frequency` have been decreased to match a good balance between performance of the system and its responsiveness.

./config/my_local_costmap_params.yaml defines properties of the costmap for the local path planner. `update_frequency` and `publish_frequency` values have been decreased. `width` and `height` are also set small. The latter was a key change to enable correct path planning in the project. Small local costmap produces correct path, which traces along passages of the labyrinth closer to the global path rather than trying to approach the goal directly across walls.

./launch/my_amcl.launch holds parameters for Particle Filter. Since Gazebo simulator represents a perfect world and does not have any noise, `odom_alpha1..4` and `initial_cov_...` parameters, described in [1] and [6] in detail, have been set very small. Such values produce more accurate localization results, but they should probably be increased for the real world.

5 Discussion

my_bot turned out to be more maneuverable and easer to navigate than udacity_bot due to its decreased weight, size and cylindrical shape.

The initial plan to put the driving wheels to the rear side of the robot failed. Such configuration did not allow the robot to rotate without linear speed motion. And a robot in-place rotation capability is an expectation from the standard `TrajectoryPlannerROS` local path-planning component.

It looks like amcl has some limited capability to restore robot localization after kidnapping. In some cases when the robot is kidnapped, ROS Particle Filter starts generating particles across the entire labyrinth, which restores robot position and orientation. Enabling and proper tuning this functionality requires further exploration of ROS parameters (`global_localization` would be a good starting point of the research).

Particle Filter, or Monte Carlo Localization has a wide applicability in the industry domain, due to its implementation simplicity and good parallelization of calculations. The limitation of a small state size can be overcome by using a compound filter, consisting of Particle Filter, and Kalman Filter, allocated for each particle (such as Rao-Blackwelization, FastSLAM and FastSLAM2 in [1]).

6 Future Work

ROS amcl implementation of Particle Filter allocates amount of particles, dependently on the performance power of the underlying platform. For faster platforms, more particles will be allocated, which will lead to more accurate and robust tracking.

In order to further improve the accuracy of the current LIDAR-based implementation of Particle Filter, sparse features, extracted from the camera and matched against the predefined and triangulated map could be additionally used. Sparse feature outliers could be filtered out by OpenCV solvePnPRansac() or similar algorithm.

Particle and Kalman Filter contain hyper-parameters, which manual tuning can be tedious and suboptimal. Automatic tuning process described in [10] and [11] can lead to more accurate results, increasing the efficiency of adaptation of such localization and tracking algorithms.

References

[1] Sebastian Thrun et al, *Probabilistic Robotics*, 2006

[2] Roger R Labbe Jr, *Kalman and Bayesian Filters in Python*, 2018, https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

[3] Cameron Davidson-Pilon, *Bayesian Methods for Hackers*, 2016, https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers

[4] Carol Fairchild et al, *ROS Robotics by Example, Second Edition*, 2017

[5] Kaiyu Zheng, *ROS Navigation Tuning Guide*, 2016

[6] http://wiki.ros.org/amcl

[7] http://wiki.ros.org/navigation

[8] http://wiki.ros.org/costmap_2d

[9] Richard Wang, *[Tutorial] Building a Simulated Model for Gazebo and ROS from Scratch,* 2016, https://www.youtube.com/watch?v=8ckSl4MbZLg

[10] Pieter Abbeel et al, *Discriminative Training of Kalman Filters*, 2005

[11] https://scikit-optimize.github.io

[12] https://en.wikipedia.org/wiki/List_of_moments_of_inertia