View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Lincoln Institutional Repository

A lightweight navigation system for mobile robots

M. T. Lázaro[1], G. Grisetti[1], L. Iocchi[1], J. P. Fentanes[2], M. Hanheide[2]
[1] DIAG, Sapienza University of Rome, Italy
[2] LCAS, University of Lincoln, UK

SPQReL Team - http://tinyurl.com/spqrel

1 Introduction

Navigation is the most important feature of mobile robot applications and is a primary functionality in RoboCup@Home, where several tests require and focus on it. RoboCup@Home teams Navigation performance of RoboCup@Home teams increased over the years thanks to the availability of standard open-source software (e.g., ROS amcl and move_base). However, with the introduction of the RoboCup@Home Social Standard Platform (i.e., SoftBank Pepper robot), a new challenge in navigation arised. Current solutions for navigation of Pepper robots have the following limitations: 1) ROS-based solutions cannot be run on the on-board PC of the robot, thus requiring network communication of sensor data that does not guarantee safety in the navigation tasks; 2) NAOqi Navigation package is not open-source and it provides only a limited set of functionalities.

In this report, we describe the development and release of an open-source lightweight navigation system that has been developed in the context of the RoboCup@Home Social Standard Platform League, within the team SPQReL that is a joint research collaboration between Sapienza University of Rome, Italy and University of Lincoln, UK.

The navigation package provided by our team overcomes the above mentioned problems since: 1) it is open-source, 2) it runs on the Pepper on-board PC, 3) it provides navigation performance comparable with ROS standard tools. Thanks to the modularity of the developed software, it can be effectively used also on ROS-based systems. As an example, we have used the ROS version on small mobile robots controlled only by a Raspberry PI3 board as replacement of amcl and move_base nodes that require much more computational resources.

2 Software components

The navigation package released contains two main components: 1) localization, 2) path planning, trajectory following and obstacle avoidance. A short description of the techniques used in these components is provided below.

2.1 Localization

The localization module implements an efficient version of the Monte Carlo Localization. The idea is to track the robot position with a particle filter. The state of the filter is the 2D robot pose $\mathbf{x}_t = (x_t; y_t; \theta_t)^T$ and the belief space is represented as a set of pose samples $\mathbf{x}^{(i)}$. The denser the samples in a region of the environment, the more likely is that the robot will be in that region. The state transition is governed by odometry measurements $\mathbf{u}_t = (\Delta x_t; \Delta y_t; \Delta\theta_t)^T$, that express the relative movement of the robot between subsequent frames. The measurements $\mathbf{z}_t$ are the laser endpoints.

Each time a new odometry measurement is received, we generate a new set of samples according to the following equation

$$\mathbf{x}^i_{t|t-1} = \mathbf{x}^{(i)}_{t-1} \oplus (\mathbf{u}_{t-1} + \mathbf{n}^{(i)}_{t-1}) \quad (1)$$

$$\mathbf{n}_{t-1} \sim \mathcal{N}(\mathbf{n}_{t-1}; \mathbf{0}, \mathbf{\Sigma}_{t-1}). \quad (2)$$

Here $\mathbf{n}_{t-1}$ is a sample drawn from a zero mean Gaussian distribution representing the additive noise affecting the odometry. The covariance of this distribution is adapted based on the magnitude of the odometry motion. If the robot does not move, then $\mathbf{\Sigma}_{t-1} = \mathbf{0}$ and no sampling is performed. If one of the samples falls in the

invalid space of the environment (e.g. unknown or inside a wall), it is replaced by a new sample drawn at a random valid location. This substantially enhances the performance during global localization.

When a measurement $\mathbf{z}_t$ becomes available, we refine our predicted belief $\{\mathbf{x}_{t|t-1}^{(i)}\}$ through conditioning. To this end we assign each predicted particle a weight $w^{(i)}$, proportional to the likelihood of the measurement. The likelihood $l(\mathbf{z}_t, \mathbf{x}) \in \Re$ is a function expressing how well the current measurement $\mathbf{z}_t$ approximates a predicted measurement obtained from the known map if the robot was at location $\mathbf{x}$. Ideally if the predicted measurement and the actual one are the same, the likelihood is maximal. Once the likelihood is computed for each predicted sample, we generate the posterior distribution by replicating or suppressing samples depending on their weight. More formally, we draw a set of new indices from the weight distributions, as follows:

$$i_t \sim w^{(i_{t-1})} \tag{3}$$

In our system we use a fast but robust procedure to calculate the likelihood. First, given a robot pose hypothesis $\mathbf{x}_t^{(i)}$ and the current laser measurement, we compute the position of the laser endpoints in the map as follows

$$\hat{z}_k^{(i)} = \mathbf{x}_{t|t-1}^{(i)} \oplus \hat{z}_k^{(i)} \tag{4}$$

where the index k represents one specific beam within the laser measurement $\mathbf{z}_t$. Subsequently, for each endpoint we compute minimal distance between the endpoint and the closest obstacle in the map. This operation can be performed in $O(1)$, by using a precalculated grid that stores for each cell the minimal distance to the obstacles: the distance map. To lessen the effects of dynamic unpredicted obstacles, we clamp the reported distance to a maximum distance value. Let $d_k^{(i)}$ be the distance of the kth beam from the closest obstacle w.r.t. particle i. If a measurement is perfectly explained, the distances will be zero. The final likelihood is computed as $exp(-\sum \sigma d^{(i)})$, where σ is a scaling factor to adjust the sensor noise.

2.2 Path planning and obstacle avoidance

Our system approaches planning by computing the minimal paths to the goal on a 2D grid using the Dijkstra algorithm. Instead of reporting just a single path, we compute a policy such that each cell of the grid points to the closest cell to reach the goal. We assume that the robot can travel from a cell to its eight neighbors, and that the cost of the transition decreases linearly with the distance from the obstacles. The cost is saturated to a maximum value. Computing the cost on a grid 1000×1000 takes 20 ms on a Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz.

The distance map is recomputed at each new measurement by incorporating the detected obstacles. This operation can be performed efficiently in an additive fashion by adding only the obstacles to the distance map representing the static scene. To track dynamic obstacles we keep a list of unexplained laser endpoints (or grid cells), that are those endpoints that fall far from an occupied cell in the static map. Obstacles are suppressed when a certain time passes or when a measurement confirms that they have been removed. The latter situation occurs when a measurement "goes through" an existing obstacle.

3 Software organization and development team

The released software is organized in a modular way as a set of C++ libraries for Linux, divided in three categories: 1) platform-independent libraries, 2) platform-dependent wrappers, 3) development tools. Platform-independent libraries include the functions for solving the main problems considered, abstracting with respect to the particular robotic development platform on which it will be integrated. These libraries are then wrapped in two platform-specific environments: ROS and NAOqi. Finally, a set of tools are released for effectively use the main library.

So the full package can be easily imported and used in ROS-based or NAOqi-based robots, but it can also easily adapted to other platforms and environments.

ROS wrappers for localization and navigation are compatible with standard ROS nodes (amcl and move_base), by using the same convention of topics, frames and parameters. So ROS nodes of our software are interchangeable with standard ROS nodes for localization and navigation, allowing for an easy replacement of such components.

3.1 Development team

The development team is formed by researchers at Sapienza University of Rome, Italy and University of Lincoln, UK who have long experience in developing and releasing software for the research communities (some examples are provided in SPQReL web site).

We will keep on working on this project before and after RoboCup 2017, maintaining it and providing assistance for other research groups and RoboCup teams that want to use the released software. Being open-source, we will be also happy to accept contributions by other groups/teams to improve the features and the performance of the components.

4 Software use

4.1 Download and installation instructions

The software is released as a git repository at https://github.com/LCAS/spqrel_navigation/. Links and additional instructions are available from the repositories Wiki (https://github.com/LCAS/spqrel_navigation/wiki and the SPQReL web site. The software is released as source code to be compiled with NAOqi, but also Debian packages for Ubuntu 16.04LTS desktop development will be provided, to ease development and use.

The current release focuses on NAOqi framework, while the ROS wrappers will be added soon in a next release.

4.2 Interfaces with other components and parameters setting

Integration with other components of the robotic applications is performed in different ways depending on the platform-dependent wrapper used. The ROS wrappers use standard ROS topics and actions for interacting with other ROS nodes, as explained in several tutorials about ROS navigation. Parameters also correspond to the equivalent ROS nodes. In the remainder of this section, we explain how to integrate SPQReL localization and navigation modules in a NAOqi environment.

The SPQReL localization and navigation components require as input a map of the environment described in the same YAML format used in ROS navigation applications. Both components run as separate processes with respect to other components and communicate through the NAOqi shared memory (ALMemory), either by means of writing and reading data directly in memory or by raising/subscribing to memory events. The main interfaces with external components that are implemented are: 1) setting the initial pose of the localizer, 2) setting navigation target pose, 3) receiving navigation status, 4) setting parameters of the components.

A brief summary of these interfaces is reported below, while more detailed information are given in the software documentation.

Set the initial pose for the localizer. The initial pose for the localizer can be set through the command line options or by clicking on the GUI provided (see Sec. 6).

Get the result of the localizer. The computed robot pose (x, y, θ) within the given map is provided as a 3-sized float vector on the ALMemory key `/NAOqiLocalizer/RobotPose`.

Set the target pose for navigation. Navigation goal can be set by clicking on the GUI provided or by raising an event on ALMemory in `NAOqiPlanner/Goal` passing as parameter the (row, col) pixel's coordinates of the goal.

Read/change the status of the navigation action. The navigation module writes in the ALMemory key `NAOqiPlanner/Path` the current path that is being executed. When the navigation towards the goal has been completed, it raises an event `NAOqiPlanner/GoalReached`. Furthermore, the navigation module subscribes to the event `NAOqiPlanner/Reset` to restore the initial status of the navigation environment by canceling the current goal and the added obstacles. It is also possible to prevent the robot from applying the computed velocities by raising the event `NAOqiPlanner/MoveEnabled` with a `true` or `false` value.

Set the parameters. Through the command line it is possible to set the parameters driving the localization filter, such as number of `particles`, `max_range`, `min_weight` or the max `distance_threshold` to compute the distance map, and of the navigation component, such as `max_cost`, `min_cost`, `robot_radius` and `safety_region` to tune the behavior of the planner.

5 Performance

Considering the use of the software on a standard platform with common computational characteristics, as is the context of the RoboCup SSPL, performance of the released localization and navigation components depends

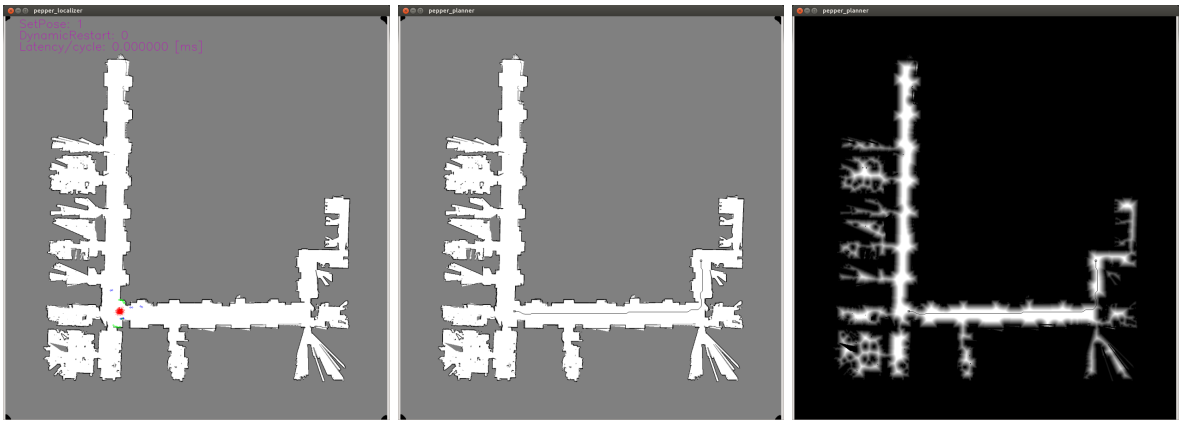

Figure 1: **Left:** Localizer viewer. Red points represent the filter particles. Green points are current laser points that can be explained by the map while blue points are laser points not explained by the map (e.g., new obstacles not represented in the map). **Middle:** Planner viewer. **Right:** Distance map.

mainly on the size of the map and its resolution. Tests realized with a map of 50×50m at a resolution of 5cm (i.e., a grid of 1000×1000, see Fig. 1) report average cycle times of 10 ms for the localization component and 100 ms for the navigation component running in the Pepper robot. Localization cycle includes: 1) reading of odometry and laser scans from ALMemory, 2) filter predict and update phases, and 3) insertion of computed robot pose in ALMemory. Navigation cycle includes: 1) reading of robot pose and laser scans from ALMemory, 2) management of obstacles in the distance map, 3) computation of cost map, 4) computation of path and its publication in ALMemory, and 5) computation of control commands (i.e., linear and angular velocities) to be applied to the robot.

6 Visualization Tools

For ROS-based systems, RViz can be used to visualize the information relevant to the task (localization particles, path planned, etc.) For non-ROS systems, we have developed simple viewers to view the information processed by the components and to tune the parameters. In particular, we will describe below the localizer viewer and the planner viewer.

Through the localizer viewer it is possible to see the outcome of the localization system. As shown in Figure 1(left), the viewer shows the map, the current laser scans and the particles that are currently stored in the filter. The viewer also allows for setting the initial pose of the robot in a specific pose of the environment or to call for a global localization phase if the pose is not known accurately.

Similarly, the planner viewer allows to visualize the state of the navigation system. As shown in Figure 1(middle), the viewer shows the map, current pose of the robot provided by the localization system, current goal (if given) and the computed path if the goal is reachable from the current pose. From this viewer it is also possible to visualize the current distance map computed from the given map and the added dynamic obstacles (see Figure 1(right)), cancel a goal or restore the distance map to its original state (i.e., cancel the obstacles added during the navigation task).

7 On-going and future work

On-going work that will be released before RoboCup2017 includes two new components for higher level planning:

1) Topological path planning: is used to generate a route plan to locations in a topological map, which indicates the robot where it should perform its tasks and which navigation actions to realize. This kind of higher level topological representation can speed up path planning, adds semantic information about navigation behaviors, and provides a more natural way of specifying tasks.

2) Petri Net Plans (PNP)[1] execution framework: is a formalism for representing high-level plans and a library for execution of such plans. The main execution engine is wrapped on different robot development environments: ROS and NAOqi. The integration of PNP with the navigation package described above allows for high-level definition of complex plans. More information is provided in the paper references in the PNP web site.

[1] pnp.dis.uniroma1.it