TU Berlin Robotics WS2019/2020

Version: 10.12.2019, 12:00

Lab Assignment #4

Please upload your solution by **Monday, January 20, 2020 at 23:59**, using your ISIS account. Remember that this is a hard deadline, extensions impossible! Uploading the solution once per group is sufficient.

This assignment mainly deals with localization from laser input of a mobile robot, given a map of its environment. We will implement a particle filter that robustly localizes an iRobot Create in a maze. All the code you'll write will take advantage of ROS (the Robot Operating System), which is a popular Message Oriented Middleware for robot systems. ROS provides network-transparent communication between nodes and for declaring connections between producers and consumers. Certain nodes provide the interface to the robot's hardware, its sensors and actuators. ROS also provides methods to replay recorded messages (ROS bags). A detailed introduction including tons of tutorials can be found at `www.ros.org`.

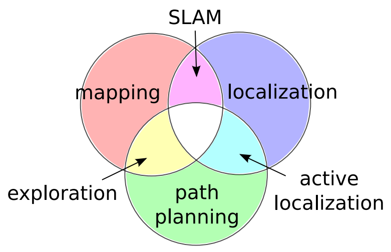

Figure 1: Diagram of a mobile robot

The assignment code is partitioned into four ROS packages:

mapping	Code template for ROS node of task B
localization	Code template for ROS node of task C
rbo_create	Interface to robot and recorded data
create_gui	ROS nodes to visualize the system state

A Bayes (5 points)

In this first little exercise, you should familiarize yourself with Bayes' rule, which relates causal with diagnostic knowledge. It is the basis of any recursive Bayesian filtering technique.

(Bayes' rule: $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$)

1. Imagine a robot that is equipped with a sensor for detecting whether a door is open or closed. The robot obtains a measurement $z = 42$ from this sensor. From previous experience, we know that $P(z = 42 \mid \text{open}) = 2/3$ and that $P(z = 42 \mid \neg\text{open}) = 1/3$. The door can only be either completely open or completely closed. We also know that the door is open more often than not, i.e. $P(\text{open}) = 3/5$.
   Compute the probability, $P(\text{open} \mid z = 42)$, of the door being open given the measurement $z = 42$ and our prior knowledge about the door.

B Mapping with Raw Odometry (35 points)

Two-dimensional evidence grids are one of the most popular representations of the environment in mobile robotics. They subdivide the environment into a grid of rectangular cells and associate with each cell an occupancy probability indicating whether the cell is occupied or not. A probability of 0 means that the corresponding cell is most likely unoccupied while a probability of 1 means that the cell is most likely occupied. There exist many methods for updating the occupancy probability of the cells as new observations become available. One of them is the *Simple Counting Method*, where the reflectance probability of a cell depends on how often a laser beam went through and how often it ended in that particular cell.

You will implement your code in the ROS package `mapping` (src/mapping/src/MappingNode.cpp)

1. Implement the *Simple Counting Method* for computing a *Reflection Map* of the environment within the ROS package `mapping`.

   Use the method stub `MappingNode::laserReceived(...)` in `mapping/src/MappingNode.cpp` to update the values of the reflection map `nav_msgs::OccupancyGrid map.`
   In the method stub, we already prepared some code that subscribes to the odometry and laser messages and that tracks the current pose of the robot given its odometry (see `odomPose`). Additionally, we have implemented two methods for your convenience:
   `MappingNode::bresenhamLine(...)` for calculating a line in a grid and
   `MappingNode::updateReflectionMapValue(...)` for updating the reflection map at a given 2D coordinate (see code for further details).

   Furthermore, the format of the laser scan message is documented at
   www.ros.org/doc/api/sensor_msgs/html/msg/LaserScan.html, the occupancy grid at
   www.ros.org/doc/api/nav_msgs/html/msg/OccupancyGrid.html.

2. Verify your results using the sensor data recorded in file `mapping/maze_corridor.bag` (in rosbag format: http://wiki.ros.org/Bags) provided with this assignment. Save three screenshots of the visualization of the map building process as shown in the `map_view`. The screenshots should be taken roughly at the time steps $t_1 \approx 2s$, $t_2 \approx 8s$ and $t_3 \approx 20s$ of the rosbag. Please include these generated images in your solution.

   Tip 1: Don't forget to build your code by invoking `catkin_make` in the workspace folder.
   Tip 2: You can use the `.launch` file for starting all necessary nodes: `roslaunch mapping mapping.launch`
   Tip 3: Playback of the rosbag can be paused with <space>.

3. Why does the resulting map differ so much from reality? Describe how the result could be improved.

C Monte Carlo Localization: Particle Filter (60 points)

Once we have a map of our environment, we can turn to the problem of localizing the robot within it using local, ambiguous sensor information. To solve this problem, we will implement a particle filter as described in the lecture. Use the map `localization/maze.yaml` and the recorded bag `localization/maze_localize.bag`. They will be launched automatically if you use `'roslaunch localization mcl.launch'` to start up.
You will implement your code in the ROS package `localization` (src/mapping/src/ParticleFilter.cpp).
We already prepared some code that subscribes to the odometry and laser data generated by the robot's sensors.
Concretely, you will need to:

1. Write a method to uniformly distribute a particle set with N particles throughout the map. Use the method stub `ParticleFilter::initParticlesUniform`. You can test your implementation by calling the ROS service `distribute_uniform`.
   Hint: Take a look at the `localization/src/Util.cpp` for helpful functions.

2. Write a method to distribute a particle set with N particles using a Gaussian distribution at pose $(\mu_x, \mu_y, \mu_\theta)$ with standard deviation $(\sigma_x, \sigma_y, \sigma_\theta)$. Use the method stub `ParticleFilter::initParticlesGaussian`. You can test your implementation by calling the ROS service `distribute_normal` with the appropriate arguments or by right-clicking in the GUI `map_view`.

3. Write a method for sampling a new pose given an old pose and an odometry command. Use the method stub `ParticleFilter::sampleMotionModelOdometry`. Note, that the parameters for the odometry-based motion model already exist (`odomAlpha1`,...). You can test your implementation by running the particle filter on the data in rosbag `maze_localize.bag`, ignoring the laser messages.
   Hint 1: The particle filter should be quite stable with the parameters given in the launch file. You don't have to tune them unless you want to study their effects.
   Hint 2: When computing the difference between two angles, make sure to handle the discontinuity at 2π sensibly (`localization/src/Util.cpp` is your friend here)

4. Write a function that, given a range measurement z_t, a pose x_t and a map m, computes the likelihood $p(z \mid x, m)$ of the measurement. Implement the Likelihood Field Model to compute the likelihood.

   (a) You should first precompute the likelihood field for a given map in the method stub `ParticleFilter::setMeasurementModelLikelihoodField`. You will need the distance transform `distMap` which is calculated in `ParticleFilter::calculateDistMap`.

   You can visualize your generated likelihood field in the map viewer by clicking on the checkbox "show likelihood field" and then "publish all".

   (b) The correction step of the particle filter should then take place in `ParticleFilter::likelihoodFieldRangeFinderModel`. When you calculate the probability of a scan don't integrate every beam as they are not independent. Rather include only every n-th beam. The member variable `ParticleFilter::laserSkip` is defined for this purpose.

   (c) Include a screenshot of the likelihood field in your report.

5. Write a method to resample the particles according to their importance weights.

   (a) Implement the stochastic universal resampling algorithm in the method stub `ParticleFilter::resample`.

   (b) Write some code to estimate the robot pose from your set of particles. Find the most likely particle and use it to update the member variable `ParticleFilter::bestHypothesis`.

6. Test your implementation in two scenarios using the rosbag `maze_localize.bag` provided with this assignment.

   (a) Global Localization: Start with a uniformly distributed particle set. Include eight screenshots showing the map with the particles, estimated robot pose and robot path in your solution. They should roughly represent the time steps $t_1 = 5s$, $t_2 = 10s$, $t_3 = 14s$, $t_4 = 20s$, $t_5 = 30s$, $t_6 = 42s$, $t_7 = 60s$ and $t_8 = 75s$ of the rosbag.

   (b) Tracking: Start with Gaussian distributed particles around the lower left corner of the maze, i.e. $\mu = (4.1,\ 6.5,\ 4.5)$ and $\sigma = (0.2,\ 0.2,\ 0.5)$. You can use the ROS service `distribute_normal` or the parameter server to initialize the filter. Generate screenshots as in (a).

Deliverables

Your solution must contain the following files:

- your **ROS workspace** directory containing the **commented source code** (excluding build/ and devel/) , especially the files:
  - `ParticleFilter.cpp`,
  - `MappingNode.cpp`
- **one** pdf file containing:
  - the commented solution for the Bayes calculation
  - Text answers
  - **Figures**. All plots must have a legend and all axes must be labeled.
  - A **table** listing for all **implementation tasks** which team member(s) implemented them (see explanation below).

Explanation and template for implementation table

- Every group member needs to be able to **answer 'high level' questions about *ALL* tasks** of the assignment.
  - We will check that during the presentations with general questions. Everyone needs to be able to answer these.
- For **implementation**, please **specify which team member** worked on which (sub-)task.
  - We will check that during the presentations with implementation specific questions.
  - You can split up implementation of (sub-)tasks. But over all, every one needs to contribute equally to the implementation. (Writing the report does *not* count as "contributing to the implementation".)
- Please use the following template in your submission:

Student Name	(A)	B1	B2	(B3)	C1	C2	C3	C4	C5	C6
Albert Albono		x	x							x
Betty Barlow						x	x	x		
Charlie Crockett						x	x	x		
Daisy Dolittle					x			x	x	x

Notes

- If building the project with `'catkin_make'` fails at first, try `'catkin_make -j1'` instead or build the packages rbo_create and localization before create_gui.
- create_gui depends on wxWidgets. We tested it with the following libraries: libwxgtk2.8-dev libwxgtk2.8-dbg
- Check http://wiki.ros.org/roscpp/Overview/Logging for logging options with ROS.
  For example: You can output debug messages using the commands `ROS_DEBUG("Hello %s", "World");` or `ROS_DEBUG_STREAM("Hello " << "World");`
  The launchfile (e.g. `'mapping/mapping.launch'`) specifies if the node's output is written to a logfile (`output=''log''`) or to the screen (`output=''screen''`).
  Be aware of different verbosity levels (debug, info, warn, error, and fatal).
  Remove all unnecessary output before submitting your solution.
- If you get the error `'ERROR: cannot launch node of type [map_server/map_server]: map_server'` you need to install the ROS package 'map_server'. You can do so with the command:
  ```
  sudo apt-get install ros-VERSION-map-server
  ```
  , where VERSION can be 'indigo', 'kinetic', or whichever ROS version you are using.

References

- Recommended Book: Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics.* MIT press, 2005.

- You can find a lot of documentation about ROS on ros.org; the following pages are directly related to understanding the compilation and use a ROS package:
http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment