Exercise 4 Mapping

Objective

The objective of this practical is that students gain experience in obtaining an environment representation using onboard sensors. Students will learn how to create a 2D gridmap using odometry and a range sensor. Moreover, the students will work on globally consistent maps.

Introduction

Representing and understanding the environment is a fundamental capability for an intelligent robot. Usually robots use an environment representation like a map in various versions (geometric or topological). In general such maps are not available in advance and need to be obtained by the robot. This is the mapping process. Due to the facts that sensor readings are uncertain and the map usually cannot be directly obtained mapping is a state estimation problem. In particular mapping is a state estimation problem with two interlinked estimation problem. Thus, mapping is generally named the *Simultaneous Mapping and Localization Problem (SLAM)* because the position or path of the robot and the map needs to be estimated simultaneously.
In this exercise we will obtain a 2D gridmap using the robot's odometry and a laser range finder (lidar). A gridmap is a grid structure of random variables that represent the probability that a particular area of the environment is occupied by an object like a wall. In order to ease the SLAM problem we will follow the *Mapping with Known Pose* approach where we assume an oracle that tells us the pose or path of the robot and estimate only the map.

Assignment

Setup

Code

In order to have the current version of the supplied code, make sure to execute the following commands before starting to work on the assignment sheet.

```
cd ~/catkin_ws/src/mobile_robots_public/
git pull
```

This task is also utilizes the simulated version of a TurtleBot 3. To get the necessary packages for the assignment, make sure to execute the following commands before starting to work on the assignment.

```
sudo apt install ros-melodic-turtlebot3 ros-melodic-turtlebot3-msgs \
    ros-melodic-turtlebot3-simulations ros-melodic-kobuki-msgs ros-melodic-csm
```

Robot

The robot used in this assignment is the TurtleBot 3 robot platform (see Figure 1). It is a differential drive robot based on its predecessor, the TurtleBot 2, which is based on the research version of the iRobot Roomba vacuum cleaning robot (iRobot Create).

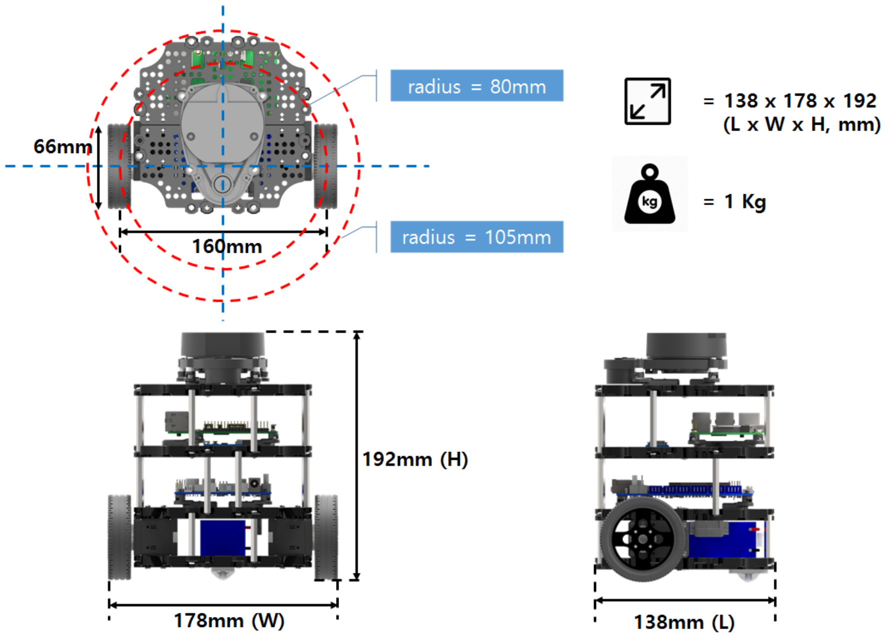

Figure 1: Dimensions of the TurtleBot 3 Burger [Image: robotis.com].

The robot is fully supported by ROS and provides a lot of sensors:

- wheel encoder / odometry
- IMU (gyroscope, accelerometer, magnetometer)
- 360° LiDAR sensor on top

The robot provides odometry information where the pose of the robot is estimated using data from internal sensors.

Simulator

The robot and its sensors will be simulated in the 3D simulator *Gazebo* [1]. The simulator simulates the motion of the robot in a 3D environment. Moreover, its simulates sensors, such as the range measurements of a 2D LiDAR to objects in the environment. The motion execution is deterministic and error-free. Thus, the estimation of the robot's pose using odometry is accurate. The robot can be controlled via the use of a tele-operation node, that allows to move the robot using the following keys of the keyboard. The keys W and X are used to set a forward and a backward linear velocity, A and D are used to set a left or right turning angular velocity and S is used to set all velocities to zero.

Visualization

The position of and the sensor data measured by the robot will be visualized using RViz [2]. This tool allows to subscribe to topics and allows to visualize the data received on those topics. This includes the visualization of different coordinate frames of the robot, laser scans, maps and more.
Using `rviz` you are able to visualize the different coordinate systems attached to the robot and their changing if the robot is moved, as well as the recorded laser scan and the generated occupancy grid map.

Software Framework

The software framework used for the assignment is based on the Robot Operation System (ROS) [3]. The setup is tested for Ubuntu 18.04 and ROS Melodic. The setup is a catkinazied ROS package. The code is available at `https://git.ist.tugraz.at/courses/mobile_robots_public`.

The setup comprises of basic packages for the real TurtleBot robot [1], the packages for the simulated robot [2] and the message and service types [3] and the own ROS node *simple_turtle* in the package *tug_simple_turtle*. The latter provides a C++ skeleton for implementing the mapping approach.
In order to start working with the robot, add the following line to your `~/.bashrc` and / or `~/.zshrc` file, depending on whether you use *bash* or *zsh*:

- `export TURTLEBOT3_MODEL=burger`

After adding this line, restart every terminal or call one of the following commands, depending on whether you use *bash* or *zsh*:

- `source ~/.bashrc`
- `source ~/.zshrc`

In order to start the robot, use the following command (in a new command line):

- `roslaunch tug_simple_turtle turtle_map.launch`

This should bring up the simulation environment *Gazebo*, visualize the currently to the robot relevant topics, published messages, the robot model and the map in *rViz* and run the `simple_turtle` node you need to modify generate the map.

Using the following command allows for controlling the robot using the keyboard (in a new command line):

- `roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch`

Task 1

The goal of the task is to implement mapping with known poses using odomery information and distance data from the recorded laser scan.

[1] `http://wiki.ros.org/turtlebot3`
[2] `http://wiki.ros.org/turtlebot3_simulations`
[3] `http://wiki.ros.org/turtlebot3_msgs`

The basic estimation problem for a map m can be defined as:

$$m^* = \arg\max_m \; [P(m|u_1, z_1, u_2, z_2, \cdots, u_{t-1}, z_{t-1}, u_t, z_t)]$$

where u represents the control inputs to the robot and z represents the recorded measurements until time t.
A grid map represents a 2 dimensional array of random variables with the probability $P(m_i = occupied)$ that a cell m_i is occupied by an object. Using the assumption that the individual cells are independent, we can reformulate the map probability in the following way:

$$P(m|u_1, z_1, \cdots, u_t, z_t) = \prod_i P(m_i|u_1, z_1, \cdots, z_{t-1}, u_t, z_t)$$

The probability $P(m_i = occupied)$ is dichotomous. So the probability can be expressed using the *log-odd* ratio that avoids problems with rounding and saturation and allows summation instead of multiplication when dealing with factorization of probabilities:

$$l(x) = log\left(\frac{P(x)}{P(\neg x)}\right) = log\left(\frac{P(x)}{1 - P(x)}\right)$$

If we assume that we know the pose x of the robot we can reformulate the map estimation problem as follows:

$$P(m_i|z_{1\ldots t}, x_{1\ldots t})$$

Using the Markov assumption and the conditioned Bayes law we can reformulate the estimation problem as follows:

$$= \frac{P(m_i|z_t, x_t)P(z_t|X_t)P(m_i|z_{t-1}, x_{t-t})}{P(m_i)P(z_t|z_{1\ldots t-1}, x_{1\cdots t})}$$

This formulation represents a recursive Bayes Filter for the map estimation problem. Given the probability of the map cell at time $t - 1$ as well as the measurement z_t and the pose x_t at time t we can estimate the probability of the map cell at time t.
We can formulate the same relation also for the counter probability:

$$P(\neg m_i|z_{1\ldots t}, x_{1\ldots t}) = \frac{P(\neg m_i|z_t, x_t)P(z_t|X_t)P(\neg m_i|z_{t-1}, x_{t-t})}{P(\neg m_i)P(z_t|z_{1\ldots t-1}, x_{1\cdots t})}$$

Using the log-odd ratio we can represent the map probability by:

$$l(m_i|z_{1\ldots t}, x_{1\ldots t}) = l(m_i|z_t, x_t) + l(m_i|z_{1\ldots t-1}, x_{1\ldots t-1}) - l(m_i)$$

This formulation represents and update rule for the log-odd ratio representation of a map cell. The first part represents the *inverse sensor model* that represents the probability of a cell being occupied given a measurement and a pose. The second part represents the log-odd ratio representation of the previous time step. The last part represents prior knowledge about the probability that a map cell is occupied. Usually this factor is zero as the probability of a cell being occupied if no information is available is assumed to be 0.5, representing uncertainty about the cell's state.

An important component here is the inverse sensor model that represents the probability that a cell is occupied given a pose and a range measurement. See Figure 2 for an example.

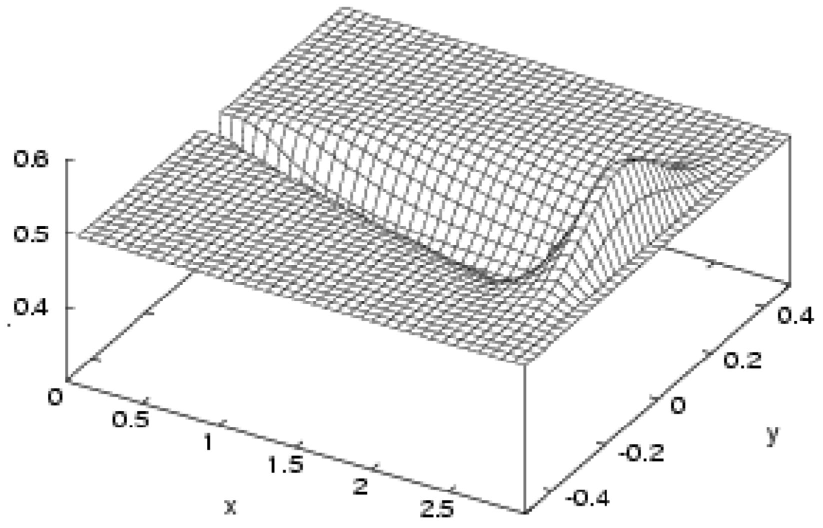

Figure 2: Inverse Sensor Model for a Range Sensor. For the original see [4].

The sensor model distinguishes basically two areas. This comprises the area that is inside the sensor cone (defined by the opening angle and maximum range) and the area outside. For the outside area the sensor provides no information, so the probability is modeled by 0.5. Within the cone we distinguish three areas. Until the measured distance that is caused by a reflection by an obstacle we can assume that the probability for a cell being occupied is low. This is represented by the valley in Figure 2. Usually the measurement uncertainty increases with the response angle we see a smooth transformation of the probability towards the uncovered area. For the area of the actual response we have a higher probability for the cell being occupied. Beyond the actual response we have again no information about the status of the cell, as sensors usually are unable to look trough obstacles. As before, we have a smooth transition between probabilities in the areas due to the uncertainties in the sensor characteristics.

Given the inverse sensor model and the recursive filter formulation we are able to integrate new sensor information in order to solve the map estimation problem. Figure 3 depicts the update of a gridmap using one range measurement.

For this, the code of the file `simple_turtle.cpp` in `~/catkin_ws_mr/src/mobile_robots_public/tug_simple_turtle/src` needs to be modified. The skeleton holds the member `occupancy_grid_map_` of type `Tug2dOccupancyGridMap`. It represents a occupancy grid map with log-odd ratio representation. It will be created with a given size (number of cells in x and y direction), a cell size in meter (squared cells) and an offset for the center of the map. It is basically organized as a double indexed array and can be quired and updated in that way. Helper functions support the access using real world coordinates. The node is organized in a way that it automatically publishes the map in the background to be visualized in `rviz`.

Every time new laser scan is received, the callback function `void`

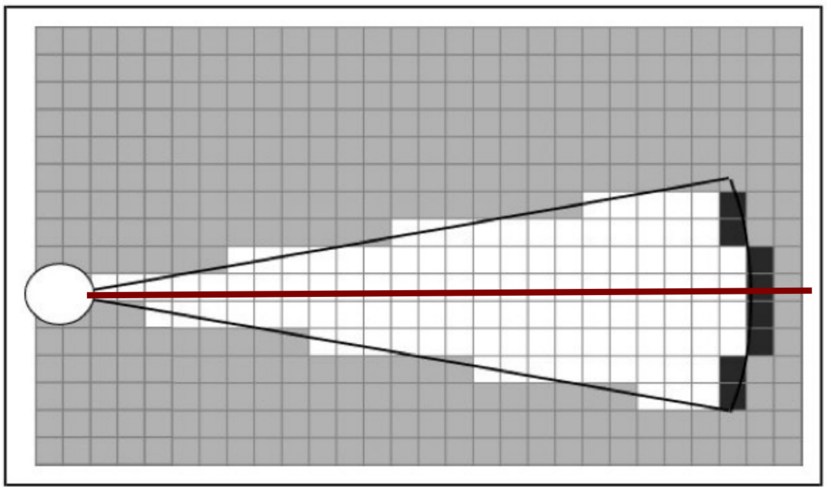

Figure 3: Gridmap after the update with a range measurement. Black denotes a most likely occupied cell. White denotes a most likely free cell. Gray denotes a cell which is not conclusive. For the original see [4].

`SimpleTurtle::laserScanCallback(const sensor_msgs::LaserScan::ConstPtr& msg)` is triggered. The sensor message `msg` contains information about the sensor characteristics, as well as the actual measured laser scan. Based on the received laser scan, you need to update the gridmap. As oracle for the actual pose of the robot you can use the odometry messages received by the node. Please consider in the map update that a laser scan comprises several individual range measurements. In order to implement an efficient update you need a kind of ray tracing along the sensor beam. It is recommended to look into the *Bresenham* algorithm for drawing lines in images.
Once the code was modified the code needs to be compiled. The command `catkin_make` in the folder `~/catkin_ws` is doing the job. In order to activate the changes the node needs to be restarted.
Please use `rviz` to visualize the mapping process. Move the robot around using the keyboard to map the entire laboratory. Please observe the reaction of your implementation to measurement errors and dynamic changes in the environment like moving obstacles or persons.

Task 2

Due to the incremental approach to integrate sensor readings into the gridmap and the uncertainty of the sensor readings, it is possible that conflicting situations like the one depicted in Figure 4 may appear. Due to the conflict, the open door in a corridor is not represented in the map correctly.
In order to improve this situation we can follow a global approach where measurements are considered simultaneously in order to maximize the probabilities of the obtained map given the sensor readings. We can reformulate the map estimation problem in the the following way:

$$m^* = \arg\max_m \left[P(m|u_1, z_1, u_2, z_2, \cdots, u_{t-1}, z_{t-1}, u_t, z_t)\right]$$

$$m^* = \arg\max_m \left[\frac{P(z_{1\ldots t}|m, x_{1\ldots t})P(m|x_{1\ldots t})}{P(z_{1\ldots t}|x_{1\ldots t})}\right]$$

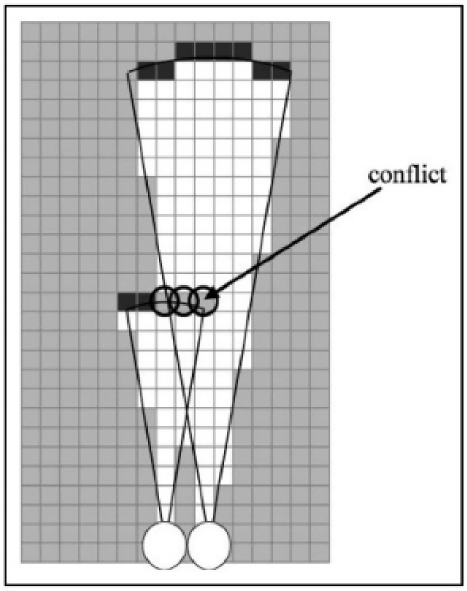

Figure 4: Conflicting gridmap after the update with two successive laser scans. For the original see [4].

Given the independence of the individual cells, the fact that the prior of the map is independent of the path the robot took and the fact that the denominator is independent of the variable we maximize, we can reformulate it in the following way:

$$= \arg\max_{m} \left[\eta \prod_{t=1}^{T} P(z_t|m, x_t) P(m) \right]$$

which can be reformulated using the logarithmic representation to the following expression:

$$= \arg\max_{m} \left[const + \sum_{t=1}^{T} logP(z_t|m, x_t) + l_0 \sum_{i} m_i \right]$$

With a fixed prior of $P(m_i) = 0.5$ we can focus on the first summation in the maximization problem.

To solve this maximization problem, we can apply hill climbing approach on the probability function. We start with an empty map; $m_i = 0$. As all cells are independent we can maximize the term

$$\arg\max_{k=0,1} \left[\sum_{t=1}^{T} logP(z_t|x_t, m \; with \; m_i = k) \right]$$

individually for each cell. We can flip individual cells from occupied to free or vice versa if it improves the probability that all measurements are generated by the underlying map with the flipped cell. We can repeat this for all cells and stop if no improvement can be achieved anymore. As hill climbing is a greedy search method we might end up in a local minima.

Submission

You have to submit a **written report** about the work done in the lab and the results achieved, which is no longer than 6 pages, in form of a PDF file. Moreover, for each task the report needs to contain:

- short description what you did to solve the given task
- derivations, formulas and code snippets you used to solve the task - it needs to be comprehensive to let one understand your solution
- drawings are always welcome and helpful
- short description of problems you faced in the lab including possible reasons and solutions

Additionally you have to submit all parts of **code** you create as a ZIP file. Please provide the full modified package inside your `~/catkin_ws/src` folder as ZIP file.

Important: Both, the PDF file and the ZIP file, need be handed in online via the course page in the TeachCenter under section *Practical*.

The question hour regarding this exercise is at 14:00 on 20.01.2021.
The deadline for the submission of this exercise is at 23:59 on 27.01.2021.

Grading

The points awarded for the different tasks are listed in Table 1.

Task	Points
Task 1	15
Task 2	10
Sum	25

Table 1: Grading: Achievable points per task.

Each exercise is worth **25 points**, summing up all four exercises to a total of 100 points. Table 2 shows the relation between grades and the percentage of total points over all four tasks.

% max. points	Grade
> 87.5	1
≤ 87.5	2
≤ 75.0	3
≤ 62.5	4
≤ 50.0	5

Table 2: Grades in relation to percentage of reachable points

References

[1] *Gazebo Simulator - ROS Integration*. http://wiki.ros.org/gazebo_ros_pkgs.

[2] *RViz*. http://wiki.ros.org/rviz.

[3] *Robot Operating System (ROS)*. http://ros.org.

[4] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN: 0262201623.

URLs