# Frontier-based Exploration—Semestral Work

## Autonomous Robotics

Tomáš Petříček

May 6, 2020

## 1 Assignment

The semestral work will be carried out individually (no teams), using a simulator instead of the real robots in the labs. Remote access to the lab computers is provided to all students (see below). The task is to design and implement a system which performs autonomous exploration of an unknown environment using a single robot.

The system will be assigned score based on the amount of empty space it covers, by comparing the occupancy grid messages published at the `/occupancy` topic to the ground-truth. Namely, the true empty cells which are marked as empty in the published occupancy grid count as positive points and the true occupied and unobservable cells which are marked as empty count as negative points. Letting $o_i$ be the occupancy values in the `/occupancy/data` array, $o_i \in \{-1, 0, 1, \ldots, 100\}$, values $0 \leq o_i < 25$ will be considered empty[1] ($-1$ means unknown).

You should use components and nodes you have implemented previously, possibly modifying and improving them as necessary. The main task will be to design and implement a brand-new ROS node for high-level planning and control of the whole exploration process. The node—executable Python script—should be located at `exploration/scripts/explorer`. An empty template is provided in the student package (see below). Alg. 1 below provides an overview of a possible implementation. An exploration experiment with such a node is shown in Fig. 1.

[1] You can set similar thresholds for planning and frontier detection as you like. This threshold will be used solely for the evaluation.

**Algorithm 1** Exploration node—overview of a possible implementation

1: Pick a frontier, if any, or a random position as a goal. ▷ `frontier.py`
2: Plan a path to the goal, or go to line 1 if there is no path. ▷ `planner.py`
3: Delegate path execution to low-level control. ▷ `path_follower`
4: Monitor execution of the plan.
5: Invoke a recovery behavior if needed. ▷ `path_follower`
6: Repeat from line 1.

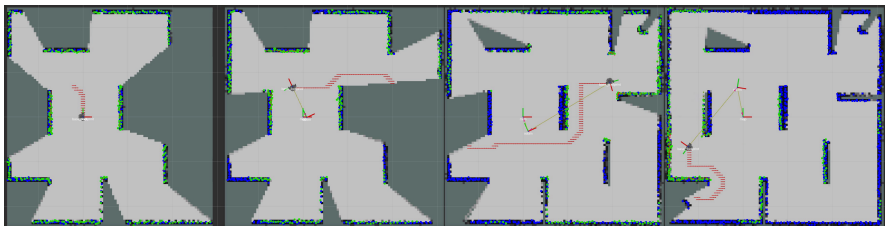

Figure 1: An exploration experiment in *stage_4*. It shows four successively planned and executed exploration paths. Each path consists of multiple poses depicted as small red arrows. Only path positions were planned and later followed, the visualized arrow orientations have no meaning. Black grid cells correspond to occupied cells, light gray to empty cells. The red, blue, and green points correspond to the measurements, the point map, and the registered points, respectively, from the `icp_slam_2d` node. See the video with the full experiment.

## 1.1 ROS Nodes

The following ROS nodes participate in exploration.

- `aro_slam/scripts/icp_slam_2d` uses (your) `aro_slam.icp` module and provides localization and mapping—including the occupancy grids which undergo evaluation.
- `exploration/scripts/frontier.py` (yours) provides services related to finding frontiers. (UPDATED: Note the small changes, e.g. adding the `robot_frame` parameter.)
- `exploration/scripts/planner.py` (yours) provides a path planning service. (UPDATED: Note the small changes, e.g. adding the `robot_frame` parameter.)
- `exploration/scripts/explorer` (yours, to be implemented) provides high-level exploration planning and control.
- `exploration/scripts/path_follower` (provided, new) provides a low-level (proportional) motion control, executing paths published by `explorer`. The node always acts upon the last received path. An empty path means stopping the robot. Without remapping it listens for `exploration/Path` messages at topic `path`.

  In the control loop, it first finds the closest point on the path and navigates towards the waypoint `look_ahead` steps ahead of this closest point. In other words, it does not maintain any history of what has been covered. As a result, having loops in the path may not work as expected—loops may be skipped or may be repeated indefinitely. The motivation was to keep the node as robust and simple as possible and to ease custom modifications, if these turn out to be necessary to handle the planned paths.

  With parameter `use_path_theta` set to `'none'`, as in the provided launch file, orientation from individual poses is neglected and the robot is oriented towards the waypoint, eventually keeping whatever orientation it

had when reaching the goal. That should be sufficient for successful exploration. Setting the parameter to `'last'` makes it use orientation of the last pose only.[2]

## 1.2 Simulation Environment

Configuration and launch files with the simulated environment are provided in separate package `aro_sim`. There are few changes in the sensor configuration from the defaults provided by `turtlebot3` packages. Namely, the source for the odometry are wheel encoders instead of ground-truth pose. As a result, the odometry must be corrected in SLAM, otherwise the localization would be inaccurate. We have also increased lidar range (to 10 m) and noise level.

Tampering with the simulation environment during evaluation is prohibited and will be penalized.

## 1.3 Evaluation

Score will be accumulated from multiple experiments, varying worlds and starting positions. The evaluation will be done on worlds similar to `stage_3`, `stage_4`, and `world`. They won't contain open areas as in `house`. A possible addition will be more maze-like with longer corridors and thick walls. Only the `burger` robot will be used in evaluation.

Please note that we will evaluate performance of the whole system in terms of the published occupancy grids, so the nodes must not only work individually but also work well with other nodes to fulfill their role in the whole system. Things to consider:

- Inaccurate localization will result in distorted maps and wrong cells being compared to the ground truth.
- Slow localization will have a negative impact on low-level motion control. Low-level motion control can be adjusted as well if needed.
- As all experiments are run in simulator, possible recovery behaviors can be quite aggressive. Nevertheless, if the maneuvers are too aggressive and robot is hitting obstacles, it will adversely affect odometry and initial pose estimates for ICP.
- Choosing inappropriate goals a visiting already covered areas repeatedly will slow down exploration.
- Having no recovery or fallback behaviors can lead the system to halt in the very beginning.

A general advice is to focus on performance bottlenecks.

More details of the evaluation procedure will be announced later.

[2] Currently, setting `'all'` is also possible (the node won't complain) but will not work without many additional changes. Besides other things, there is an interplay with the `look_ahead` parameter.

# 2 Remote Access to Labs

Students can use lab computers (E-130, E-132, E:230) remotely, including remote desktop. We suggest using remote desktop via Xpra—see the brief how-to at the course website.

If needed, more information can be found in

- the video tutorials (in Czech and English) and
- the user guide (only Czech).

# 3 Workspace Configuration

The simulation environment has been tested on Ubuntu 18.04 with ROS Melodic, using both a generic Ubuntu installation and the provided Singularity image below. Other configurations will not be supported.

## 3.1 Robolab Singularity Image

This Singularity image (UPDATED: April 22 2020) is used at the labs—it already provides all the dependencies. To enter the shell within the Singularity image:

```
singularity shell /path/to/robolab_melodic.simg
```

In the labs, there is also an expanded image available for convenience:

```
singularity shell /opt/singularity/robolab/melodic
```

To build your customized image, use the scripts from the Robolab deploy repository.

## 3.2 Catkin Workspace

The workspace can then be configured as follow:

```
ws=~/workspace/aro
mkdir -p "${ws}/src"
cd "${ws}/src"
curl -sSL https://cw.fel.cvut.cz/b192/_media/courses/aro/tutorials/aro_sim.zip -O
unzip *.zip
cd "${ws}"
catkin init
catkin config --extend /opt/ros/aro
catkin config --cmake-args -DCMAKE_BUILD_TYPE=Release
catkin build -c
```

## 3.3 Extras for a Generic Ubuntu

On a generic Ubuntu with ROS Melodic (not with the Singularity image above), dependencies must be install prior `caktin build -c` from the accompanied rosinstall file and via rosdep.

```
cd "${ws}/src"
wstool init
wstool merge aro_slam/dependencies.rosinstall
wstool up -j 4
cd "${ws}"
rosdep install --rosdistro melodic --from-paths src --ignore-src -r -y
```

Also note that we will use Gazebo 9.13 during evaluation instead of Gazebo 9 from the default Ubuntu repository. Gazebo 9.13 is provided by the Open Source Robotics Foundation and may be installed using the installation script from the robolab deploy repository.

# 4 Package Usage

Don't forget to source your workspace:

```
source "${ws}/devel/setup.bash"
```

To run the whole system inside the simulated environment:

```
roslaunch aro_sim turtlebot3.launch world:=stage_4
```

Worlds worth trying are `stage_3`, `stage_4`, `world`, and `house`.

All your nodes should be configured for evaluation within launch file `exploration/launch/exploration.launch` which is included from launch file `aro_sim/launch/turtlebot3.launch` above.