Autonomous Mobile Robots Assignment 2

Group 8

March 2022

Map Generation (Binary Occupancy Map)

The Binary Occupancy Map is a simple Two-dimensional map that contains information about where the robot can and can't go. The map is later used to create the Probabilistic Road Map.

To create the map, the environment that the map should reflect is measured and drawn in a graphics editor such that 50 pixels in the map represent 1 meter in real life. For the space, we chose a section of the hallway in the Shannon building with the lockers. It does not have the most complex geometry but it will make do!

Before the image can be used as a map, it is converted to a gray-scale image, then to a boolean matrix, which then in turn can be used with the binaryOccupancyMap constructor to get a binaryOccupancyMap instance. The map's origin is then shifted with an offset so that the place where the robot is located at and the start of its path coincide with the origin. The map is then "inflated" to compensate for the size of the robot since the robot is viewed as a singular point from an algorithmic point of view. A code snippet showing this process is included below.

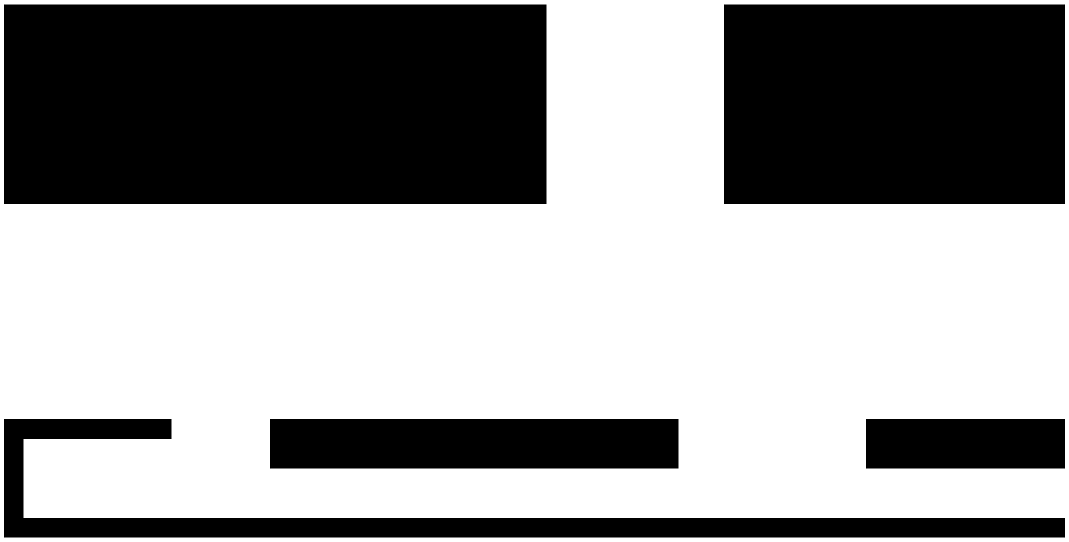

Figure 1: Initial drawing of the Shannon map

```
% Load binary occupancy map
img = imread("shannon_hallway.png");
grayImg = rgb2gray(img);     % Convert to gray scale
bwImg = grayImg < 0.5;       % Convert to black and white image

% Offset the map so origo is moved to where we want it
xOffset = -30;
yOffset = -30;
```

```
resolution = 50; % 50 pixels per meter


% Create binary occupancy map
map = binaryOccupancyMap(bwImg, resolution);
map.GridOriginInLocal = [xOffset, yOffset] / resolution;
```

```
% Inflate map to compensate for robot size (18cm is a little over the radius of the robot)
inflate(map, 0.18);
```

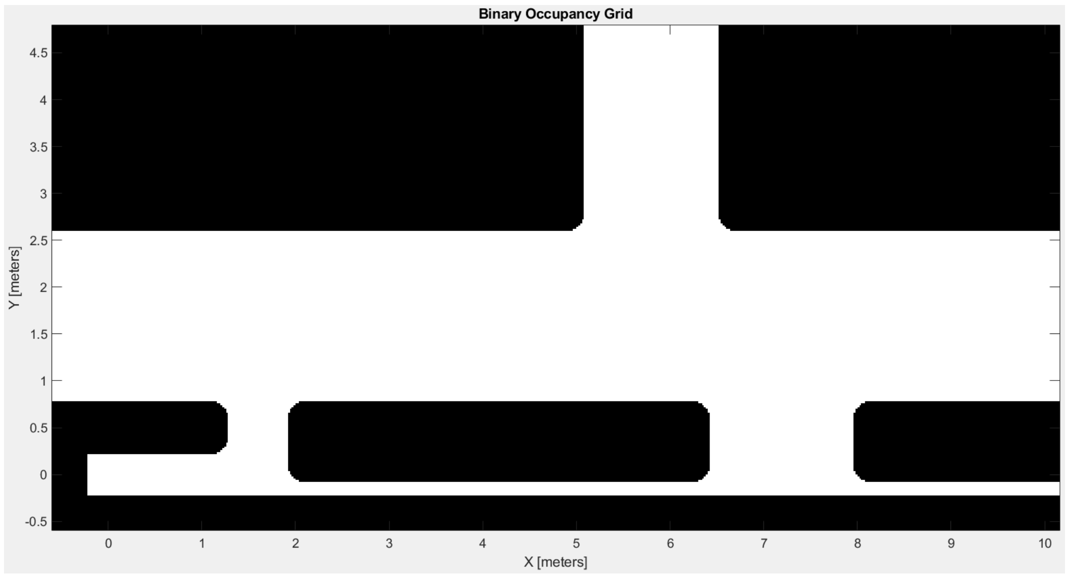

Figure 2: Final Binary Occupancy Map

Notice how over-inflating might make some crooks and passages inaccessible. Here, this is not the case.

Probabilistic Road Map

The probabilistic road map is generated based on the PRM algorithm, which takes a Binary Occupancy Map and a set of parameters. The algorithm then places a specified number of random nodes which all lie on a free space. The nodes are then connected based on the distance between a pair of nodes and whether an occupied space intercepts them. The Probabilistic Road Map can the be tuned to perform optimally in the given environment. Finally, a path is generated from the start point to the goal using a linear combination of node connections. The following code snippet show the generation of the Probabilistic Road Map.

```
% Make Probabilistic Road Map with 1000 points
prmMap = mobileRobotPRM(map, 1000);

% Define start and goal points
startPoint = [0 0];
goal = [5.7 3.87];

% Use find path function on the PRM map
path = findpath(prmMap,startPoint,goal);
```

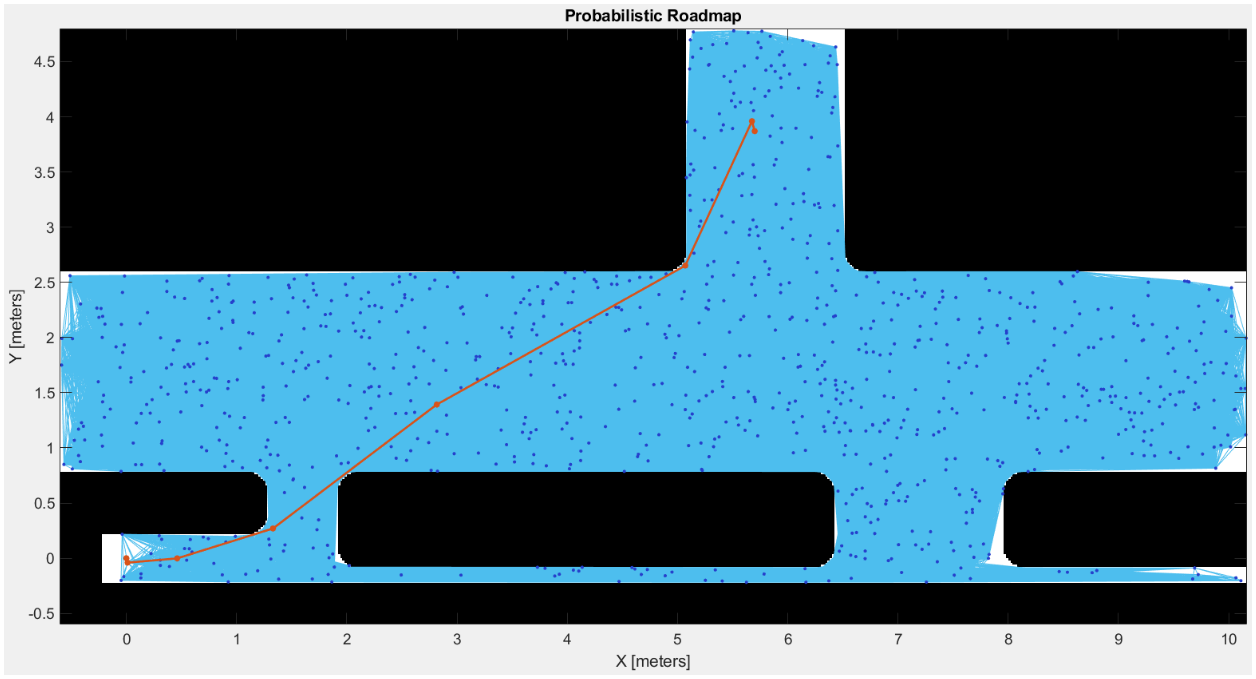

Figure 3: Probabilistic road map with path

Notice how there are both the point `startPoint` and `goal` as well as points from the PRM that are closest to those two points. This can cause a slight jerk motion at the start and end of the path, however this should be fine as long a we use a large enough look-ahead distance for the pure pursuit controller.

Pure Pursuit

The Pure Pursuit controller uses the pure pursuit algorithm to follow the path created using the probabilistic road map. The controller is fed the nodes of the probabilistic road map path along with parameters detailing desired linear velocity, maximum angular velocity and lookahead distance.

The algorithm works by following a point that travels along the generated path. The point will be ahead by the lookahead distance. A too small lookahead distance might make the robot swerve and oscillate when trying to stay consistent with the path, but a too large one will have the robot undershoot the waypoints. There lies much potential for performance improvements of the robot by simply tuning the lookahead distance and angular speeds, which can make the robot follow the path nearly perfectly if the right values are found.

```
% Initialize Pure Pursuit controller
ppCtl = controllerPurePursuit;

% Set Pure Pursuit controller parameters
ppCtl.Waypoints = path;
ppCtl.DesiredLinearVelocity = 0.25;
ppCtl.MaxAngularVelocity = 2;
ppCtl.LookaheadDistance = 0.3;
```

The proper ROS publisher/subscriber relationships is set up to enable extracting the pose of the robot to give to the pure pursuit controller as well as publishing velocity commands to the robot. The pose is, in this robots case, a three columned matrix containing an x and y coordinate along with angular position.

```
% Get ROS odometry subscriber and velocity publisher
robotOdomSub = rossubscriber('/odom');
robotVelPub = rospublisher('/mobile_base/commands/velocity');
```

```
velMsg = rosmessage(robotVelPub);

% Reset odometry of the robot
robotResetOdomPub = rospublisher('/mobile_base/commands/reset_odometry');
send(robotResetOdomPub, rosmessage(robotResetOdomPub));

timer = rateControl(60); % 60 Hz timer for the loop
goalReachedDist = 0.1;
running = true;
```

When running the robot, the pose is extracted from the odometry subscriber and the current rotation is converted from a quaternion to euler angles, from which the Y rotation is used. This, along with the current position of the robot, is then fed to the pure pursuit controller in form of the pose, which in turn provides the correct linear and angular velocities, so as to continue along the desired path. This information is then published to the robot using the '`mobile_base/commands/velocity`' topic. This execution loop then runs until the goal is reached.

```
% Execution loop; loop time determined by rate controller timer
while(running)
    % Get pose from odometry subscriber
    odomMsg = receive(robotOdomSub);

    % Convert pose orientation to Euler angles
    pquat = odomMsg.Pose.Pose.Orientation;
    quat = [pquat.W, pquat.X, pquat.Y, pquat.Z];
    angles = quat2eul(quat);

    % Save current pose to global variable
    pose = [odomMsg.Pose.Pose.Position.X
            odomMsg.Pose.Pose.Position.Y
            angles(1)];

    % Use the Pure Pursuit controller to get linear and angular velocities
    [v, omega] = ppCtl(pose);
    velMsg.Linear.X = v;
    velMsg.Angular.Z = omega;

    % Use the publisher to send the velocities
    send(robotVelPub,velMsg);

    waitfor(timer);

    % Check if goal is reached
    distToGoal = norm(goal - pose(1:2)');
    if(distToGoal < goalReachedDist)
        fprintf('Goal reached!');
        running = false;
    end
end
```

Test Results

In the appendices is included a video of the robot driving from the start point to the goal as intended.

In our script, the condition for stopping movement is checking if the robot's position is within a distance from the goal. In our test, we set this to 0.1 meters. The robot ended up being about 0.2 meters away from the goal point when it stopped, but since it was coded to stop a 0.1 meters before, we can say that the robot undershot by about 0.1 meters, which seems largely acceptable.

Making the goal reached condition smaller might cause the robot to miss the goal completely. Thus we think that it is expected to always be slightly off, unless we were to implement a way for it to properly hone in on the goal.

Another reason for the higher distance to the goal can be the initial position and orientation, as well as errors in the odometry from the robot. This combination of errors can lead to an incorrect position and orientation and be the cause for the offset.

Appendices

With the report is included the MATLAB script, the first draft of the Shannon hallway with measurements, and a video of the test on the Turtlebot.