EE4308
Turtlebot Project

Advances in Intelligent Systems and Robotics
Academic Year 2016-2017

Preben Jensen Hoel - Paul-Edouard Sarlin
A0158996B - A0153124U

March 2, 2017

Abstract

In this report we describe how we move a robot from a start point to a goal point through a unknown maze with the help of a depth camera. We also explore the possibility of dynamically changing the goal position. The path planning is done with a modified A* algorithm and everything is simulated using ROS and Gazebo. The robot used is a Turtlebot and the depth camera a Kinect camera.

Contents

1 Introduction

The purpose of this project is to design a complete and generic navigation system for a mobile robot, such that it is able to autonomously and safely navigate in a known or unknown environment, given arbitrary initial and goal positions. The robot, a Turtlebot (Figure 1), carries several sensors and is simulated in a virtual 3D maze, representing a room and composed of orthogonal walls organised along square cells (Figure 2).

We will first give an overview of our solution system, explaining the methodology that led us to design it, before closely examining each module that it is composed of. Examples of realistic situations as well as a brief analysis of our results will then be discussed.

Figure 1: The Turtlebot robot. [1]

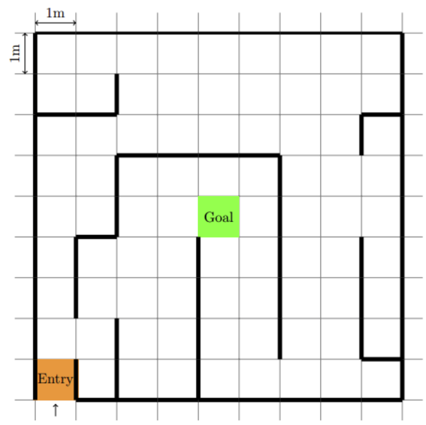

Figure 2: Example of a simple maze.

2 Overview of the proposed solution

The navigation problem can be expressed as finding the appropriate velocity commands to be sent to the robot based on data received from the environment or on *a priori* knowledge, such that it reaches a given goal position starting from an initial one. Conceptually, the robot should find a suitable and elegant trajectory in the maze, allowing to both avoiding obstacles and eventually reaching the goal within a reasonable time. Although this process may seem straightforward when executed by a human being, it is in fact a complex and multi-level behaviour. We divide such a problem in several sub-systems, that can be more easily handled and solved by a computer system.

The first step is to process the data obtained from the sensors and to deduce some useful information. In our case, motor encoders provide wheels rotation angles which can be used to deduce the position of the robot in the working area – called Odometry, while a Kinect sensor returns depth information over a specific field of view and is used for obstacles detection. As the robot's position can already be obtained from the simulation, we focus on extracting information on obstacles – wall positions in our case – from the Kinect (Section 4). The newly detected walls can be compared and added to a map stored by the robot, whether it is beforehand given or constructed by previous iterations of the currently described process.

Complete or partial knowledge of the map allows to compute a path – a list of waypoints – from the start to the goal. In the latter case, the path may not be optimal as there might be walls standing across it, leading it to be computed again when they are detected by the robot – called dynamic planning. Otherwise, the path is optimal if the right algorithm is used. In addition to path finding, some post-processing, including densifying and smoothing, may be applied to the path. These operations are all together part of the *global planner* (Section 5).

Once the final path has been generated, we have to make sure that the robot actually follows it. This is the role of the *local planner* (Section 6). Taking the current position as well as the path to follow, it computes the linear and angular velocity commands to be sent to the robot. A closed loop controller is implemented on the distance and the orientation errors relative to the next waypoints. As the given path is discrete, a more complex control approach can be used to ensure that the movement looks smooth and natural.

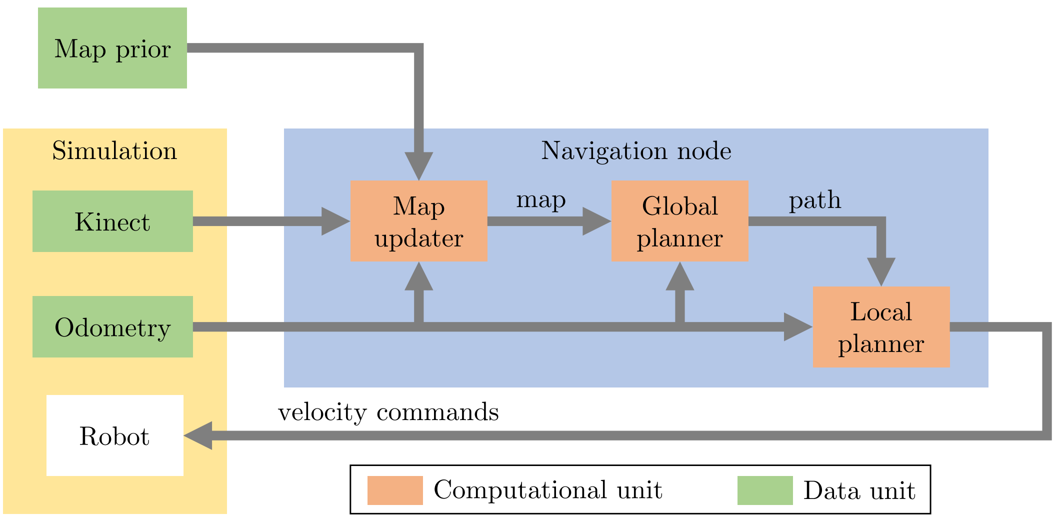

Figure 3: Overview of the navigation system.

The velocity commands can finally be sent to the robot, completing one iteration of the navigation scheme, and thus closing the loop. The process is described in Figure 3, using the above-mentioned denomination. Before taking a closer look at the different sub-systems, let us first address the implementation question.

3 Implementation

The navigation system has been designed as a package to be integrated in the Robot Operating System (ROS), a flexible and widely-used software framework. It makes it easy to interface with existing libraries and tools, such as Gazebo, a powerful simulation software which provides a realistic 3D environment for the Turtlebot. Although ROS already includes a generic navigation stack [2], it does not perform very well compared to solutions that are tailor-made for a particular and defined environment, as it is the case here. For the sake of clarity, our sub-systems however follow the same denomination as this software stack.

Our navigation node is written in Python, which has been preferred over C++ for its compactness and its prototyping-oriented syntax. It allows a more flexible and efficient data processing code for paths, maps or point clouds, in a similar way as Matlab does. Each module described hereafter corresponds to a self-contained file (see Appendix A) that can be easily reused for other applications.

The top level file is `navigation.py`. It listens to incoming data from the simulation or the user, through several callback functions associated to topics `/odom_true` – ground-truth Odometry, `/camera/depth/points` – Kinect data, and `/move_base_simple/goal` – a user-defined goal. The different sub-systems are then called depending on the action to be carried out. It is to note that, as the callbacks may be called concurrently, mutual exclusion procedures are used to ensure thread safety. An additional node, `odom_true.py`, simultaneously publishes the ground-truth Odometry from Gazebo, allowing more accurate wall detection and movement control.

4 Information gathering

4.1 Map topology

As mentioned previously, the maze has a predefined shape: it is a 9 by 9 grid of 1-meter-long square cells, and a wall can only be found at the boundary between two cells. Rather than storing the map as an image (Figure 2), we develop a custom data structure that is memory-efficient and simple to interpret and process.

We denote as integer coordinates the centre of the cells, e.g. the start cell is $(0, 0)$, while the one above is $(0, 1)$. The map is stored as a one-dimensional array containing the coordinate points of the centre of the walls. As such, horizontal walls coordinates are of form $(n, m + 0.5)$ $(n, m \in \mathbb{N})$, while vertical ones are expressed as $(n + 0.5, m)$. Since the size of the maze is known, the walls of the outer boundaries are not included in the map, but still taken into account by the planning algorithms. Let us note that, thanks to the genericness of our solution, the size could be arbitrarily changed without any effect on the performance of the navigation system.

4.2 Wall detection and map update

Walls around the robot can be detected using the Kinect sensor, which returns depth information in a field of view of 120° [3] in front of the robot. The sensor encodes the information into a point cloud, a 640 × 480 2D array of 3D points, which are defined by their X, Y, and Z coordinates in the frame of the robot. Due to the compute-intensive nature of the hereafter described wall extraction algorithm, the point cloud is only processed at a frequency of 10 Hz. As the speed of the robot is rather limited, this is enough to ensure a successful obstacles avoidance.

Since all the walls have the same height, the depth information is redundant in the vertical direction, and all the 480 rows approximately contain the same XY points – although lower rows are impacted by the ground plane. We therefore decide to extract and process a single row of points, located at half of the total height, such that it encompasses all the walls that are within the FOV. The coordinates of the resulting points are then transformed from the robot frame into the world reference frame using the known absolute position and orientation of the robot.

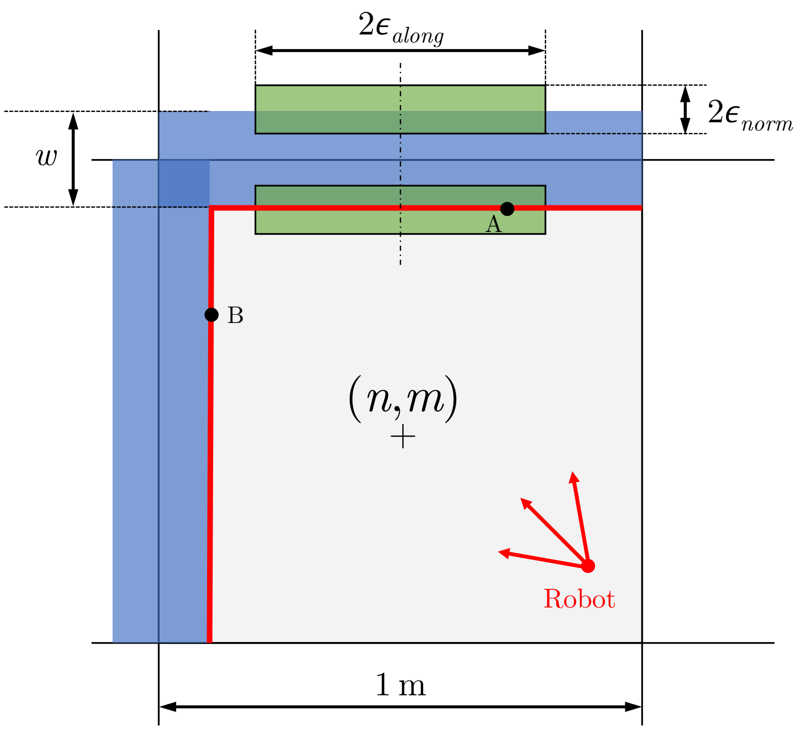

Figure 4: Matching points with potential walls.

The actual wall extraction algorithm tries to match each point to a potential wall by checking if it belongs to a region of interest (ROI). Given that a wall has a thickness of w in the simulation, we can express the points on the longitudinal surfaces of horizontal and vertical walls as respectively $(n + \Delta, m + 0.5 \pm w/2)$ and $(n + 0.5 \pm w/2, m + \Delta)$, where $\Delta \in [-0.5; 0.5]$. Figure 4 shows an example with two walls coloured in blue and points returned by the Kinect as a red line. Since the two walls overlap at their common end, points near the boundary of two cells could also correspond to transverse surfaces. As a consequence, we restrict the ROI to the central part of a wall, defined by its width $2\epsilon_{along}$. Additionally, a tolerance ϵ_{norm} in the normal direction of the actual longitudinal surface of the wall is added to deal with uncertainties in the placement of the walls as well as potential noise in the depth measurement. The resulting ROI is shown in green and is mathematically defined for horizontal and vertical walls as respectively:

$$(n \pm \epsilon_{along}, m + 0.5 \pm w/2 \pm \epsilon_{norm}) \quad \text{and} \quad (n + 0.5 \pm w/2 \pm \epsilon_{norm}, m \pm \epsilon_{along}) \tag{1}$$

Each point is compared to its closest ROI and, in case of a match, is considered as upholding the hypothesis that there is a wall at this particular location. For example, point A would be found to belong to the wall $(n, m + 0.5)$, but point B would not. A wall is only added to the map if a sufficient number of points is found to belong to it. This limits the detection distance, but makes the extraction more robust. The threshold is defined empirically, and a value of 30 has been eventually selected.

5 Global planning

The path is computed by the global planner, first finding an approximate path, which is then improved by applying several post-processing techniques.

5.1 Path finding

The famous A-Star algorithm [4] turned out to be a perfect solution for the path finding task, being both optimal and computationally efficient. It first labels each cell (x, y) with a unique index $i = y \cdot \text{width}_{\text{map}} + x$, interpreting it as the node of a graph, and then propagates a move cost from the start point. The algorithm always expands first the path that minimizes the total semi-estimated cost f from start to goal. For each cell i, $f(i) = g(i) + h(i)$, where g is the movement cost from start to i, and h is the estimated cost from i to the goal, called the heuristic function and chosen to be the Manhattan distance.

The implementation is largely inspired by [5], but adapted to the previously-mentioned custom map topology, as obstacles – walls – are here not included in the initial graph, but rather taken into account during the propagation using the array of walls. Additionally, a new parameter allows to encourage straight paths over successive sharp turns, or conversely. This is done by dynamically changing the move cost from one cell to the other by looking at the current virtual orientation of the robot. This does not affect the optimality of the final path, but rather provides more flexibility when multiple optimal solutions exist. A similar strategy encourages the first move to be in the same direction as the initial orientation of the robot. As we will see later, theses techniques allow to further improve the smoothness of the movement.

5.2 Global Smoothing

The path outputted by A* contains sharp turns that force the robot to stop and turn at each corner (red in Figure 5). In order to maximize the speed of the robot along the path, we apply a least-squares smoothing regularization that takes into account the whole path, hence called global. Let (x_i, y_i) the n points of the A* path, and $(x_i^{'}, y_i^{'})$ the ones of the smoothed path, computed such that they minimize the cost function:

$$J = \frac{1}{2}\sum_{i=1}^{n} \alpha \underbrace{\left(\left(x_i - x_i^{'}\right)^2 + \left(y_i - y_i^{'}\right)^2\right)}_{\text{original path}} + (1-\alpha) \underbrace{\left(\left(x_i^{'} - x_{i+1}^{'}\right)^2 + \left(y_i^{'} - y_{i+1}^{'}\right)^2\right)}_{\text{shortened smooth path}} \tag{2}$$

The parameter $\alpha \in [0, 1]$ expresses the trade-off between closeness to the original path and smoothness. We use gradient descent to minimize J, with the constraint that the start and goal points remain unchanged, i.e. $(x_1, y_1) = (x_1^{'}, y_1^{'})$ and $(x_n, y_n) = (x_n^{'}, y_n^{'})$.

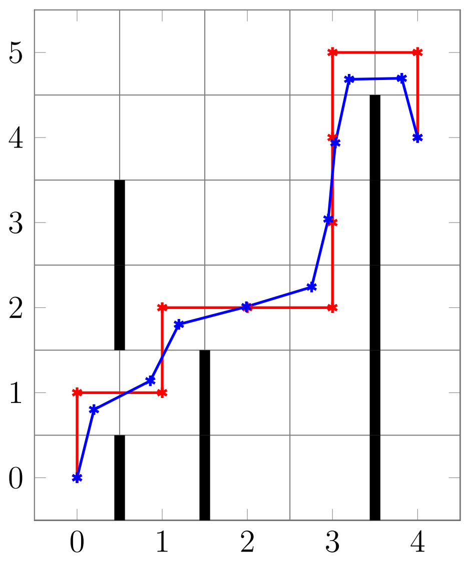

Figure 5: Outputs paths of A-Star (red) and simple global smoothing (blue) for a test scenario.

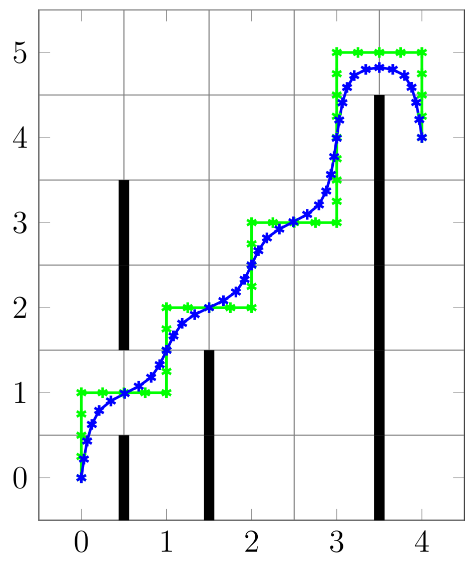

Figure 6: A-Star with maximization of turns (green) and dense global smoothing (blue), same scenario.

The blue path of Figure 5 corresponds to the result for a test scenario. It is indeed smoother than the red one, but still contains sharp turns. We decide to tune the A* path by maximizing the number of turns, and to make it denser by adding three new points on each segment, resulting in the green path of Figure 6. Applying the previous global smoothing results in the blue path, which seems much nicer than the previous one.

5.3 Dynamic path planning

The execution context of the global planner depends wether the map is beforehand known or not. If a complete and accurate prior is given, then the path planning needs to be executed only at the node startup, assuming that the simulation environment is static, i.e. walls are not moving. In that case, the Kinect processing is disabled in order to save computational power.

If there is no map prior, or if it is assumed to be only partial, the global planner needs to recompute the path each time that a new wall is detected. As such, the map updater module triggers the global planner if new walls are added to the map after a Kinect processing iteration. As mentioned earlier, the orientation of the robot tends to remain preserved during this process, avoiding unnecessary and time-consuming course corrections. The local planner is then reset if the path has changed, i.e. if the new path is not a subset of the previous one.

6 Local Planning

Ensuring that the robot's actual movement follows the computed path is the function of the local planner.

6.1 Low level control

For each new Odometry message, the linear and angular velocity commands are computed and sent to the robot. A Proportional-Integral (PI) feedback controller is used on the distance and the orientation to the next point of the path. If the orientation error is too high, the robot might significantly diverge from the path. In that particular case, the linear velocity is set to zero, so that the robot can first turn towards the next point, and then move to it. Transition from one point to the other happens when the distance error is lower than a predefined tolerance.

The speed of the robot is bounded to physically reasonable values in order to keep a realistic behaviour and to give enough time to the Kinect processor to detect walls before any collision. As this may give rise to integral wind-up, the integral part is reset for each new waypoint.

6.2 Local path simplification

To make the controller more robust and to ensure that missing a point will not produce any unnatural behaviour, the next waypoint is determined each time that the controller routine is called. First, starting from the previous waypoint, the point i closest to the current robot's position is selected. Next, the angle between the robot's position, this point i and the following one $i+1$ is computed. If it is smaller than $\pi/2$, then $i+1$ should be the next waypoint. Otherwise, i will be selected. **Add figure from powerpoint.**

This technique proves to be particularly useful when using dynamic path planning, as the newly computed global path starts from the closest cell center, which might differ from the robot's current position. The path thus might not be locally optimal — although it globally is. Cleverly selecting the next waypoints solves this problem.

6.3 Local smoothing

Although the path outputted by the global planner module is already globally smooth and simplified, the above-described control process produces a jerky movement. We thus apply a second smoothing layer, using only the next few points of the path, hence called local. Instead of using the orientation and distance error of the next point only, we decide to take into account k points by computing the weighted average of their errors. Let θ_{PI} and d_{PI} be the orientation and distance errors to be fed into the PI controller, θ_i and d_i the errors from the current position to the point i, $W = (w_i)$ a vector of k weights, and j the index of the current point:

$$\theta_{PI} = \frac{\sum_{i=1}^{k} \theta_{j+i}\, d_{j+i}\, w_i}{\sum_{i=1}^{k} d_{j+i}\, w_i} \quad \text{and} \quad d_{PI} = \frac{\sum_{i=1}^{k} d_{j+i}\, w_i}{\sum_{i=1}^{k} w_i} \tag{3}$$

As the robot gets closer to the next point, the corresponding distance error decreases, giving less importance to this point in the orientation correction: the robot smoothly turns towards the next one. After experimental trial, we find that $k = 4$ and $W = \begin{bmatrix} 7 & 4 & 2 & 1 \end{bmatrix}$ give good results. The first two points have thus more influence on the controller than the others, although the following two help to smooth the transition from one current point to the other. It is important to notice that the overall influence of the k points is strongly related to their distance apart and is thus somewhat indirectly coupled with the global smoothing point density.

7 RViz

We have implemented support for using RViz as a way to show our path and map. To make this possible we create the map based on the walls we already know. In addition to displaying the map we have also added the path and the Point Cloud, which makes it easy to see where the robot is going and what it sees. One problem we haven't been able to solve with RViz is that the map is flickering in the view, what's causing this is still unknown to us and fixing this have been deemed to time consuming to solve. **<Should we just remove the previous sentence? Up to you>** In Figure 7 we see how all this comes to life and gives us a easy to understand view of what the robot is doing. This not only provides important information and a nice graphical way to show where the robot is moving but also gives a good tool for debugging and error searching as well as a tool for optimizing the algorithm. **<Should we write more here or just delete it?>**

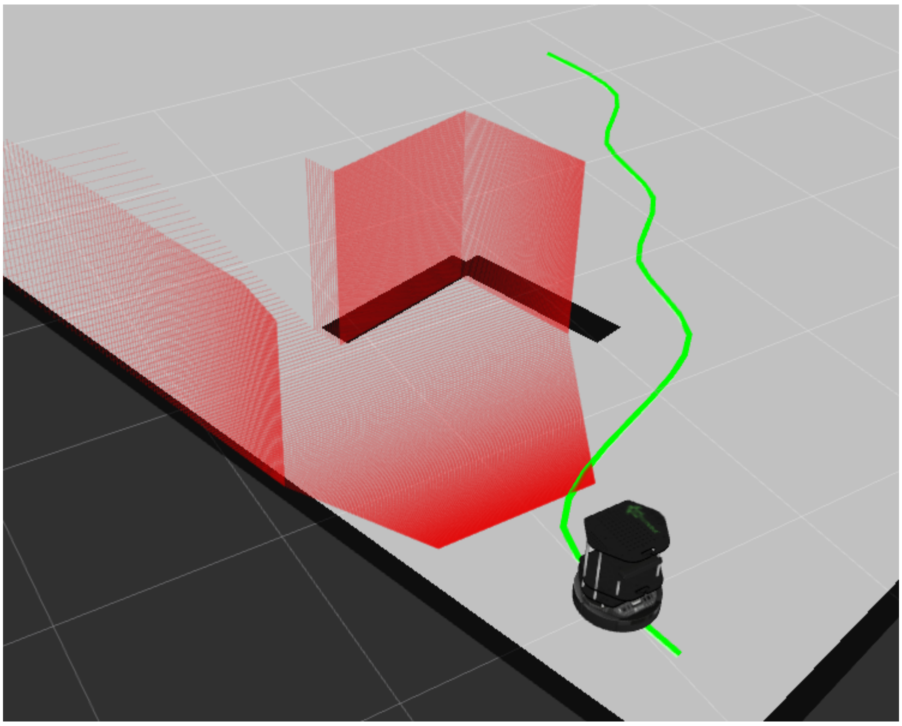

Figure 7: RViz displaying our current data, with Point Cloud in red and the path in green

In addition to giving us a nice tool for displaying our data, RViz provides us with a few other features that have been useful in testing, understanding and give our robot some extra features. The most important feature is the ability to interact with the robot in real time, this means that we can set new goal points and the robot update itself and find a path to the new goal. This together with the rest of the code have given our robot the ability not only to find it's way through a maze, but adapt to both to a changing environment and changing demands from the user.

8 Conclusion

We have seen how our robot finds it way through a maze by breaking the problems into smaller parts that each play an important role in solving the overall problem. We have also seen how we have used A* to find an optimal path and tuned the algorithm to fit our needs, while the local and global smoothing helps giving the robot a natural movement. In addition we have given the robot the ability to see it's environment through the use of a Kinect and update its target in real time with the use of RViz.

<Next sentence might be good as a "punch line" but might also backfire a bit and make our report less formal?> We mean that by doing this we have not only solved the problem at hand but also moved our robot one step closer to being able to work as a autonomous robot in real life scenarios. **<if we want this sentence then maybe build up under which tasks, these should then be stated in the intro/abstract, e.g. "These scenarios might include delivering packages, cleaning a room or more complex tasks like search and rescue">**

References

[1] http://www.turtlebot.com/. [The Turtlebot official website].

[2] http://wiki.ros.org/navigation. [Official page of the ROS Navigation package].

[3] http://smeenk.com/kinect-field-of-view-comparison/. [Kinect specifications].

[4] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[5] http://www.redblobgames.com/pathfinding/a-star/implementation.html. [Python implementation of the A-Star algorithm].

A Listings

A.1 Configuration file

```
#!/usr/bin/env python

# Title:        EE4308 Turtlebot Project
# File:         config.py
# Date:         2017-02-13
# Author:       Preben Jensen Hoel (A0158996B) and Paul-Edouard Sarlin (A0153124U)
# Description:  Configuration file containing static parameters used by several
↪   modules, for map building, path planning, or low-level control.


from math import radians as rad


# MAP PARAMETERS
KNOWN_MAP = False
GOAL_DEFAULT = (4,4)
MAP_WIDTH   = 9
MAP_HEIGHT  = 9
MAP =   [(0.5,0), #begin vertical walls
         (0.5,2),
         (0.5,3),
         (1.5,0),
         (1.5,1),
         (1.5,4),
         (1.5,5),
         (1.5,7),
         (3.5,0),
         (3.5,1),
         (3.5,2),
         (3.5,3),
         (5.5,1),
         (5.5,2),
         (5.5,3),
         (5.5,4),
         (5.5,5),
         (7.5,1),
         (7.5,2),
         (7.5,3),
         (7.5,6),
```

```
         (0,6.5), #begin horizontal walls
         (1,3.5),
         (1,6.5),
         (2,5.5),
         (3,5.5),
         (4,5.5),
         (5,5.5),
         (8,0.5),
         (8,6.5)]

X_OFFSET = 0.5
Y_OFFSET = 0.5

RESOLUTION = 0.1
WALL_THICKNESS = 0.2

ORIGIN_X = 0
ORIGIN_Y = 0
ORIGIN_Z = 0


# MAP BUILDING PARAMETERS
TOL_NORMAL = 0.1
TOL_ALONG = 0.20
TOL_NB_PTS = 30


# PATH FINDING PARAMETERS
COST_NORMAL = 1
COST_LOWER = 0.95
COST_MOVE = COST_NORMAL
COST_TURN = COST_LOWER


# SMOOTHING PARAMETERS
LOCAL_SMOOTHING  = True
SMOOTH_NB_PTS   = 4
SMOOTH_WEIGHTS  = [7, 4, 2, 1]

GLOBAL_SMOOTHING = True
SMOOTHING_TOL = 1E-6
ALPHA = 0.2
SMOOTHING_RATE = 1
```

```
SMOOTHING_DENSITY = 4


# CONTROL PARAMETERS
TOL_ORIENT  = 0.01
TOL_DIST    = 0.1
THR_ORIENT  = rad(45)
K_P_ORIENT  = 0.9
K_P_DIST    = 0.4
K_I_ORIENT  = 5e-4
K_I_DIST    = 1e-3

MAX_V_LIN = .3
MAX_V_ANG = 2
```

A.2 Navigation

```
#!/usr/bin/env python

# Title:        EE4308 Turtlebot Project
# File:         navigation.py
# Date:         2017-02-13
# Author:       Preben Jensen Hoel (A0158996B) and Paul-Edouard Sarlin (A0153124U)
# Description:  Main node of the navigation scheme. Handles communication with other
↪    ROS components and manages computational units such as local and global planners
↪    or Kinect processor.


import rospy
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, PoseStamped
from sensor_msgs.msg import PointCloud2
from tf.transformations import euler_from_quaternion
from threading import Lock
from math import cos, sin

from local_planner import LocalPlanner
from global_planner import AStar as pathSearch, globalSmoothing
from map_updater import processPcl
from rviz_interface import RvizInterface
import config as cfg
```

```python


path_raw = None
path = None
goal = None
map_updated = []
pose = None
ERROR = False

# In rospy callbacks can be called in different threads
controller_lock = Lock()
pose_lock = Lock()
map_lock = Lock()
goal_lock = Lock()
path_lock = Lock()


# Update the velocity commands and publish path to RViz
def updateController(odom_msg):
    global pose, init
    cmd = Twist()

    with pose_lock:
        pose = extractPose(odom_msg)
    if ERROR:
        (v_lin, v_ang) = (0,0)
        rospy.loginfo("Stopping robot.")
    else:
        if not init:
            setGoal(cfg.GOAL_DEFAULT)
            init = True
        with controller_lock:
            (v_lin, v_ang) = controller.update(pose)
    cmd.linear.x = v_lin
    cmd.angular.z = v_ang
    pub.publish(cmd)
    with path_lock:
        visualisation.publishPath(path)


# Wrapper as a callback
def newGoal(goal_msg):
    # Extract goal positionin frame Odom
```

```
    X = goal_msg.pose.position.x
    Y = goal_msg.pose.position.y
    # Convert to Gazebo world frame
    with pose_lock:
        x = pose[0] + cfg.X_OFFSET + X*cos(pose[2]) - Y*sin(pose[2])
        y = pose[1] + cfg.Y_OFFSET + Y*cos(pose[2]) + X*sin(pose[2])
    if (x < 0) or (x >= cfg.MAP_WIDTH) or (y < 0) or (y >= cfg.MAP_HEIGHT):
        rospy.logerr("Goal is out of the working area.")
        return
    setGoal((int(round(x - cfg.X_OFFSET)),int(round(y - cfg.Y_OFFSET))))
    with path_lock:
        visualisation.publishPath(path)


# Sets a new goal and initialize the path
def setGoal(goal_local):
    global goal
    with goal_lock:
        goal = goal_local
    rospy.loginfo("New goal set: %s", goal_local)
    computePath()


# Process Kinect data, update map accordingly
def updateMap(pcl_msg):
    global map_updated
    with pose_lock:
        pose_local = pose # processPcl might take some time, avoid blocking
          ↪ updateController
    detected_walls = processPcl(pcl_msg, pose_local)
    new_walls = [w for w in detected_walls if w not in map_updated]
    if len(new_walls) == 0:
        return
    rospy.loginfo("Discovered new walls: %s", new_walls)
    new_map = map_updated + new_walls
    with map_lock:
        map_updated = new_map
    if not cfg.KNOWN_MAP:
        rospy.loginfo("Map updated, compute new path.")
        computePath()
    visualisation.publishMap(new_map)
```

```python
def computePath():
    global path, path_raw, ERROR
    # Create local copies for path, pose, goal
    with path_lock:
        path_astar_last = path_raw
    with pose_lock:
        start = (int(round(pose[0])), int(round(pose[1])))
        theta = pose[2]
    with goal_lock:
        goal_local = goal
    # Compute AStar and smoothed paths
    try:
        if cfg.KNOWN_MAP:
            path_astar = pathSearch(start, goal_local, cfg.MAP, theta)
        else:
            with map_lock:
                path_astar = pathSearch(start, goal_local, map_updated, theta)
    except ValueError as err:
        ERROR = True
        rospy.logerr("%s", err)
        return
    else:
        ERROR = False
    if cfg.GLOBAL_SMOOTHING:
        path_final = globalSmoothing(path_astar)
    else:
        path_final = path_astar
    # Update if path changed
    if (path_astar_last is None) or (path_astar[-1] != path_astar_last[-1]) or \
       (not (set(path_astar) <= set(path_astar_last))) :
        with path_lock:
            path = path_final
            path_raw = path_astar
        with controller_lock:
            controller.reset(path_final)
        rospy.loginfo("Reset controller with new path.")
    else:
        rospy.loginfo("Keep same path, no obstacle on path.")

# Extract relevant state variables from an Odometry message
def extractPose(odom_msg):
    quaternion = (odom_msg.pose.pose.orientation.x,
                  odom_msg.pose.pose.orientation.y,
```

```
                    odom_msg.pose.pose.orientation.z,
                    odom_msg.pose.pose.orientation.w)
    euler = euler_from_quaternion(quaternion)
    theta = euler[2]
    pos_x = odom_msg.pose.pose.position.x - cfg.X_OFFSET
    pos_y = odom_msg.pose.pose.position.y - cfg.Y_OFFSET
    return (pos_x, pos_y, theta)


if __name__ == "__main__":
    global pub, controller, visualisation, init
    rospy.init_node("navigation", anonymous=True)

    if rospy.has_param("known_map"):
        cfg.KNOWN_MAP = rospy.get_param("known_map")

    rospy.Subscriber("/odom_true", Odometry, updateController)
    rospy.Subscriber("/move_base_simple/goal", PoseStamped, newGoal)
    if rospy.get_param("use_kinect", default=True):
        rospy.Subscriber("/camera/depth/points_throttle", PointCloud2, updateMap)
    pub = rospy.Publisher("/cmd_vel_mux/input/teleop", Twist, queue_size=1)

    controller = LocalPlanner()
    visualisation = RvizInterface()
    if cfg.KNOWN_MAP:
        visualisation.publishMap(cfg.MAP)
    init = False

    try:
        rospy.spin()
    except rospy.ROSInterruptException:
        rospy.loginfo("Shutting down node: %s", rospy.get_name())
```

A.3 Global planner

```
#!/usr/bin/env python

# Title:        EE4308 Turtlebot Project
# File:         global_planner.py
# Date:         2017-02-13
# Author:       Preben Jensen Hoel (A0158996B) and Paul-Edouard Sarlin (A0153124U)
```

```
# Description:  Global path planner, including path finding, densifying and
↪   smoothing.


import heapq
from math import pi, atan2, radians as rad
import config as cfg


# Useful queue class for AStar
class PriorityQueue:
    def __init__(self):
        self.elements = []

    def empty(self):
        return len(self.elements) == 0

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get(self):
        return heapq.heappop(self.elements)[1]

# Get the coordinates of a node of the graph
def getPt(idx):
    x = idx % cfg.MAP_WIDTH
    y = int(idx / cfg.MAP_WIDTH)
    return (x, y)

# Get the graph index of a grid cell
def getIdx(pt):
    (x, y) = pt
    return (y * cfg.MAP_WIDTH + x)

# Heuristic function used by AStar
def heuristic(a, b):
    (x1, y1) = a
    (x2, y2) = b
    return abs(x1 - x2) + abs(y1 - y2)

# Backtrack from the goal to build the path using the list of nodes
def buildPath(came_from, goal):
    path = []
```

```
    current = getIdx(goal)
    while current is not None:
        path.insert(0, getPt(current))
        current = came_from[current]
    return path

# AStar algorithm that finds the shortest path start and goal
def AStar(start, goal, map, theta=0):
    frontier = PriorityQueue()
    frontier.put(getIdx((start)), 0)
    came_from = {}
    cost_so_far = {}
    came_from[getIdx(start)] = None
    cost_so_far[getIdx(start)] = 0

    while not frontier.empty():
        current = frontier.get() # Get node with lowest priority
        if current == getIdx(goal):
            return buildPath(came_from, goal)

        # Determine reachable nodes
        (x, y) = getPt(current)
        neighbors = [(x + 1, y), (x, y - 1), (x - 1, y), (x, y + 1)]
        rem = []
        for pt in neighbors:
            (xp, yp) = pt
            if (xp < 0) or (xp >= cfg.MAP_WIDTH) or (yp < 0) or (yp >=
            ↪  cfg.MAP_HEIGHT) or ((( x + xp) 2., (y + yp) / 2.) in map):
                rem.append(pt)
        neighbors = [getIdx(pt) for pt in neighbors if pt not in rem]

        for next in neighbors:
            # Checks if turning or straight path is turning, assign corresponding
            ↪  weights
            if current != getIdx(start):
                (x_n, y_n) = getPt(next)
                (x_p, y_p) = getPt(came_from[current])
                if (x_n != x_p) and (y_n != y_p):
                    move_cost = cfg.COST_TURN
                else:
                    move_cost = cfg.COST_MOVE
            else:
                (x_n, y_n) = getPt(next)
```

```
                err_theta = checkAngle(atan2(y_n - start[1], x_n - start[0]) - theta)
                if abs(err_theta) < rad(45):
                    move_cost = cfg.COST_LOWER
                else:
                    move_cost = cfg.COST_NORMAL
            # Compute the movement cost from the start (g)
            new_cost = cost_so_far[current] + move_cost
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                # Compute the total cost from start to goal through this node (f)
                priority = new_cost + heuristic(goal, getPt(next))
                frontier.put(next, priority)
                came_from[next] = current
    raise ValueError('Goal '+str(goal)+' cannot be reached from '+str(start))

# Global smoothing taking into account all the points
def globalSmoothing(path):
    # Densify the path by adding points
    dense = []
    for i in range(0, len(path) - 1):
        for d in range(0, cfg.SMOOTHING_DENSITY):
            pt = []
            for j in range(0, len(path[0])):
                pt.append(((cfg.SMOOTHING_DENSITY - d) * path[i][j] + d * path[i + 1][j]) float(cfg.SMOOTHING_DENSITY))
            dense.append(tuple(pt))
    dense.append(path[len(path) - 1])

    smoothed = [list(pt) for pt in dense] # convert from tuple to list
    err = cfg.SMOOTHING_TOL

    # Minimizes the cost function using gradient descent
    while err >= cfg.SMOOTHING_TOL:
        err = 0
        for i in range(1, len(dense) - 1):
            for j in range(0, len(dense[0])):
                tmp = smoothed[i][j]
                smoothed[i][j] = smoothed[i][j] + \
                    cfg.SMOOTHING_RATE * (cfg.ALPHA * (dense[i][j] - smoothed[i][j]) +
                    (1 - cfg.ALPHA) * (smoothed[i + 1][j] +
                    smoothed[i - 1][j] - 2. * smoothed[i][j]))
                err = err + abs(tmp - smoothed[i][j])
```

```
    return smoothed

def checkAngle(theta):
    if theta > pi:
        return(theta - 2 * pi)
    if theta < -pi:
        return(theta + 2 * pi)
    else:
        return(theta)
```

A.4 Local planner

```
#!/usr/bin/env python

# Title:        EE4308 Turtlebot Project
# File:         local_planner.py
# Date:         2017-02-13
# Author:       Preben Jensen Hoel (A0158996B) and Paul-Edouard Sarlin (A0153124U)
# Description:  Local path planner performing velocity commands computations with
 ↪  path following and smoothing.


import rospy
from math import atan2, sqrt, pow, pi, copysign
import config as cfg


class CtrlStates:
    Orient, Move, Wait = range(3)


class LocalPlanner:
    # Reset the controller, find nearest start point
    def reset(self, path):
        self.ctrl_state = CtrlStates.Move
        self.sum_theta = 0.
        self.sum_dist = 0.
        self.path = path
        self.pts_cnt = 0

    # Compute first target point of the path from current position
```

```
    def findNext(self, pose):
        position = (pose[0],pose[1])
        cnt = self.pts_cnt
        dist_best = self.dist(self.path[0], position)
        for i in range(cnt,len(self.path)):
            dist = self.dist(self.path[i], position)
            if dist > dist_best:
                break
            else:
                dist_best = dist
                cnt = i
        if i != (len(self.path)-1):
            dist_next = self.dist(self.path[i+1], position)
            dist_inter = self.dist(self.path[i+1], self.path[i])
            if dist_next < dist_inter:
                cnt +=  1
        self.pts_cnt = cnt

    # Update the PI controller, including local smoothing
    def update(self, pose):
        (pos_x, pos_y , theta) = pose

        while True:
            v_lin = 0.
            v_ang = 0.

            if self.ctrl_state == CtrlStates.Wait:
                return (v_lin, v_ang)

            self.findNext(pose)

            # Check if apply local smoothing or used global one
            if cfg.LOCAL_SMOOTHING:
                nb_err_pts = cfg.SMOOTH_NB_PTS
            else:
                nb_err_pts = 1

            # Compute orientation and distance error for the next points
            err_theta = []
            err_dist = []
            for (x, y) in self.path[self.pts_cnt:self.pts_cnt + nb_err_pts]:
                err_theta_raw = atan2(y - pos_y, x - pos_x) - theta
                err_theta.append(self.checkAngle(err_theta_raw))
```

```
                err_dist.append(sqrt(pow(x - pos_x, 2) + pow(y - pos_y, 2)))

            if cfg.LOCAL_SMOOTHING:
                (err_dist_tot, err_theta_tot) = self.weighted_errors(err_dist, err_theta)
            else:
                err_theta_tot = err_theta[0]
                err_dist_tot = err_dist[0]

            self.sum_theta += err_theta_tot
            self.sum_dist += err_dist_tot

            # Control intial orentation
            if self.ctrl_state == CtrlStates.Orient :
                # Check if satisfactory => start motion
                if (abs(err_theta_tot) < cfg.TOL_ORIENT) or (err_dist[0] < cfg.TOL_DIST) :
                    self.ctrl_state = CtrlStates.Move
                    self.sum_theta = 0.
                    self.sum_dist = 0.
                    continue
                # Else adjust orientation
                else:
                    v_ang = cfg.K_P_ORIENT * err_theta_tot + cfg.K_I_ORIENT * self.sum_theta
                    break

            # Control motion between waypoints
            elif self.ctrl_state == CtrlStates.Move :
                # Check if satisfactory => next point
                if err_dist[0] < cfg.TOL_DIST :
                    if self.pts_cnt == (len(self.path) - 1):
                        self.ctrl_state = CtrlStates.Wait
                        break
                    else:
                        self.pts_cnt += 1
                        self.sum_theta = 0.
                        self.sum_dist = 0.
                        continue
                else:
                    if (abs(err_theta_tot) > cfg.THR_ORIENT):
                        rospy.loginfo("Orientation error is too big, turning...")
                        self.ctrl_state = CtrlStates.Orient
```

```python
                        self.sum_theta = 0.
                        self.sum_dist = 0.
                        continue
                    else:
                        v_ang = cfg.K_P_ORIENT * err_theta_tot + cfg.K_I_ORIENT * self.sum_theta
                        v_lin = cfg.K_P_DIST * err_dist_tot + cfg.K_I_DIST * self.sum_dist
                        break
        return self.checkVelocities(v_lin, v_ang)

    # Compute weigthed distance and orientation errors in local smoothing
    def weighted_errors(self, err_dist, err_theta):
        err_theta_tot = 0
        err_theta_scaling = 0
        err_dist_tot = 0
        err_dist_scaling = 0
        for i in range(len(err_theta)):
            err_theta_tot += err_theta[i] * err_dist[i] * cfg.SMOOTH_WEIGHTS[i]
            err_theta_scaling += err_dist[i] * cfg.SMOOTH_WEIGHTS[i]
            err_dist_tot += err_dist[i] * cfg.SMOOTH_WEIGHTS[i]
            err_dist_scaling += cfg.SMOOTH_WEIGHTS[i]
        err_theta_tot /= err_theta_scaling
        err_dist_tot /= err_dist_scaling
        return (err_dist_tot, err_theta_tot)

    # Euclidian distance
    def dist(self, p1,p2):
        return sqrt(pow(p1[0] - p2[0], 2) + pow(p1[1] - p2[1], 2))

    # Enforce velocities limits
    def checkVelocities(self, v_lin, v_ang):
        return(copysign(min(abs(v_lin),cfg.MAX_V_LIN),v_lin), copysign(min(abs(v_ang),cfg.MAX_V_ANG), v_ang))

    # Convert the angle to [-pi,pi]
    def checkAngle(self, theta):
        if theta > pi:
            return(theta - 2 * pi)
        if theta < -pi:
            return(theta + 2 * pi)
        else:
            return(theta)
```

A.5 Map updater

```python
#!/usr/bin/env python

# Title:        EE4308 Turtlebot Project
# File:         map_updater.py
# Date:         2017-02-13
# Author:       Preben Jensen Hoel (A0158996B) and Paul-Edouard Sarlin (A0153124U)
# Description:  Kinect processor extracting wall coordinates from the 2D point cloud.


import rospy
from sensor_msgs.msg import PointCloud2
from sensor_msgs import point_cloud2 as pcl2
from math import cos, sin
import config as cfg


def processPcl(pcl_msg, pose):
    h = pcl_msg.height
    w = pcl_msg.width
    # Isolate ROI from Pcl
    roi = zip(range(w),[int(h/2)]*w)
    pcl = pcl2.read_points(pcl_msg, field_names=("x", "y", "z"), skip_nans=True, uvs=roi)
    # Extract coordinates in world frame
    pcl_global = toGlobalFrame(pcl, pose)
    new_walls = extractWalls(pcl_global)
    return new_walls

# Convert from robot frame to global world frame
def toGlobalFrame(pcl, pose):
    pcl_global = []
    for (y,z,x) in pcl: # curious order...
        y = -y # Y axis is inverted in received message
        X = pose[0] + x*cos(pose[2]) - y*sin(pose[2])
        Y = pose[1] + y*cos(pose[2]) + x*sin(pose[2])
        pcl_global.append((X,Y))
    return pcl_global
```

```python
# Find walls from point cloud
def extractWalls(pcl):
    candidates = []
    for (x,y) in pcl:
        err_norm_x = abs(x - round(x - .5) - .5) - cfg.WALL_THICKNESS/2
        err_norm_y = abs(y - round(y - .5) - .5) - cfg.WALL_THICKNESS/2

        # Check if could be a vertical wall
        if abs(err_norm_x) < cfg.TOL_NORMAL:
            spread_y = y - round(y)
            if abs(spread_y) < cfg.TOL_ALONG:
                x_wall = round(x - 0.5) + 0.5
                y_wall = round(y)
                if (x_wall <= -0.5) or (x_wall >= (cfg.MAP_WIDTH-0.5)) or (y_wall < 0) or (y_wall >= cfg.MAP_HEIGHT):
                    continue
                candidates = addPoint(candidates, x_wall, y_wall)

        # Or a horizontal one
        elif abs(err_norm_y) < cfg.TOL_NORMAL:
            spread_x = x - round(x)
            if abs(spread_x) < cfg.TOL_ALONG:
                x_wall = round(x)
                y_wall = round(y - 0.5) + 0.5
                if (y_wall <= -0.5) or (y_wall >= (cfg.MAP_HEIGHT-0.5)) or (x_wall < 0) or (x_wall >= cfg.MAP_WIDTH):
                    continue
                candidates = addPoint(candidates, x_wall, y_wall)
    # Filter candidates with enough points
    detected_walls = [(x,y) for (x,y,cnt) in candidates if (cnt >= cfg.TOL_NB_PTS)]
    return detected_walls

# Take into account a new wall
def addPoint(candidates, x_wall, y_wall):
    already_detected = False
    for i in range(len(candidates)):
        if (x_wall, y_wall) == (candidates[i][0], candidates[i][1]):
            candidates[i][2] += 1 # increase number of corresponding points
            already_detected = True
            break
    if not already_detected:
        candidates.append([x_wall, y_wall, 1])
    return candidates
```

A.6 Odometry

```
#!/usr/bin/env python

# Title:        EE4308 Turtlebot Project
# File:         odom_true.py
# Date:         2017-02-13
# Author:       Preben Jensen Hoel (A0158996B) and Paul-Edouard Sarlin (A0153124U)
# Description:  Publishes the ground truth state of the robot acquired from Gazebo.


import rospy
import tf
from gazebo_msgs.msg import ModelStates
from nav_msgs.msg import Odometry
from geometry_msgs.msg import PoseWithCovariance, Pose, TwistWithCovariance, Twist,
↪ TransformStamped


robot_name = "mobile_base"


def callback(model_states):
    rospy.logdebug("Received ModelStates")
    try:
        idx = model_states.name.index(robot_name)
    except ValueError:
        rospy.logerr("[ModelStates] Could not find model with name %s", robot_name)
        return

    model_pose = model_states.pose[idx]
    model_twist = model_states.twist[idx]

    msg = Odometry()
    msg.pose.pose = model_pose
    msg.twist.twist = model_twist
    pub_odom.publish(msg)

    t2.header.stamp = rospy.Time.now()
    t2.transform.translation = model_pose.position
    t2.transform.rotation = model_pose.orientation
```

```
    tfm2 = tf.msg.tfMessage([t2])
    pub_tf.publish(tfm2)
    rospy.logdebug("Published Tf.")


def initialize():
    global pub_odom, pub_tf, t, t2
    rospy.init_node("odom_correcter", anonymous=True)
    rospy.Subscriber("/gazebo/model_states", ModelStates, callback)
    pub_odom = rospy.Publisher("/odom_true", Odometry, queue_size=1)
    pub_tf = rospy.Publisher("/tf", tf.msg.tfMessage, queue_size=10, latch=True)

    #Transformation for the world frame
    t = TransformStamped()
    t.header.frame_id = "world"
    t.header.stamp = rospy.Time.now()
    t.child_frame_id = "/dummy_link"
    t.transform.translation.x = 0.0
    t.transform.translation.y = 0.0
    t.transform.translation.z = 0.0
    t.transform.rotation.x = 0.0
    t.transform.rotation.y = 0.0
    t.transform.rotation.z = 0.0
    t.transform.rotation.w = 1.0

    t2 = TransformStamped()
    t2.header.frame_id = "world"
    t2.child_frame_id = "base_footprint"

    rospy.spin()


if __name__ == "__main__":
    initialize()
```

A.7 RViz

```
#!/usr/bin/env python

# Title:         EE4308 Turtlebot Project
# File:          rviz_interface.py
```

```python
# Date:         2017-02-13
# Author:       Preben Jensen Hoel (A0158996B) and Paul-Edouard Sarlin (A0153124U)
# Description:  Publishes the map and path as standard ROS messages to be displayed in Rviz.


import rospy
from nav_msgs.msg import OccupancyGrid, Path
from geometry_msgs.msg import PoseStamped
from math import floor
import config as cfg


class RvizInterface:
    def __init__(self):
        self.pub_map = rospy.Publisher("/map", OccupancyGrid, queue_size=1, latch=True)
        self.pub_path = rospy.Publisher("/path", Path, queue_size=1, latch=True)

        self.map = OccupancyGrid()
        self.map.header.frame_id = "world"
        self.map.info.resolution = cfg.RESOLUTION
        self.map.info.width = int(cfg.MAP_WIDTH / cfg.RESOLUTION)
        self.map.info.height = int(cfg.MAP_HEIGHT / cfg.RESOLUTION)
        self.map.info.origin.position.x = cfg.ORIGIN_X
        self.map.info.origin.position.y = cfg.ORIGIN_Y
        self.map.info.origin.position.z = cfg.ORIGIN_Z

        self.path = Path()
        self.path.header.frame_id = "world"

    # Contruct and publish the path message
    def publishPath(self, path):
        self.path.poses[:] = []
        for i in range(len(path)):
            p = PoseStamped()
            p.pose.position.x = path[i][0] + cfg.X_OFFSET
            p.pose.position.y = path[i][1] + cfg.Y_OFFSET
            p.pose.position.z = 0
            self.path.poses.append(p)
        self.pub_path.publish(self.path)

    # Contruct and publish the map message (Occupancy Grid)
```

```
    def publishMap(self, walls):
            # Initialize 2D map with zeros
            map = []
            for i in range(self.map.info.height):
                row = []
                for j in range(self.map.info.width):
                    row.append(0)
                map.append(row)

            # Add border
            for i in range(self.map.info.height):
                map[i][0] = 100
                map[i][self.map.info.width - 1] = 100
            for j in range(self.map.info.width):
                map[0][j] = 100
                map[self.map.info.height - 1][j] = 100

            # Iterate through the walls, set the pixels accordingly
            for (x, y) in walls:
                if (y % 1) == 0:
                    # Wall is vertical
                    y += cfg.Y_OFFSET
                    x += cfg.X_OFFSET
                    for i in range(int(1 / cfg.RESOLUTION)):
                        map[int(x / cfg.RESOLUTION)][int((y - cfg.Y_OFFSET) / cfg.RESOLUTION + i)] = 100
                        map[int(x / cfg.RESOLUTION - 1)][int((y - cfg.Y_OFFSET) / cfg.RESOLUTION + i)] = 100
                else:
                    # Wall is horizontal
                    y += cfg.Y_OFFSET
                    x += cfg.X_OFFSET
                    for i in range(int(1 / cfg.RESOLUTION)):
                        map[int((x - cfg.X_OFFSET) / cfg.RESOLUTION + i)][int(y / cfg.RESOLUTION)] = 100
                        map[int((x - cfg.X_OFFSET) / cfg.RESOLUTION + i)][int(y / cfg.RESOLUTION - 1)] = 100

            self.map.data = []
            # Flatten map to self.map.data in a row-major order, publish
            for i in range(len(map)):
                for j in range(len(map[0])):
                    self.map.data.append(map[j][i])
```

```
                self.pub_map.publish(self.map)
```