Computing Science

CS4048 Robotics Assessment Report

Hasan Bayarov Ahmedov

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: January 8, 2021

CS4048 Robotics Assessment Report

Department of Computing Science
University of Aberdeen
King's College
Aberdeen AB24 3UE

January 2021

CS4048 Robotics Assessment Report

Hasan Bayarov Ahmedov

Department of Computing Science
University of Aberdeen

January 8, 2021

1 Introduction

This assessment evaluated our understanding of robotics through the use of ROS and RViz to visualise a vacuum cleaner robot and the environment. Also, it tested our grasp of path planning algorithms along with robot localisation, control and image processing. The entire assessment took place within the ROS Stage simulator, where the robotic vacuum cleaner was implemented and was driven to different locations where vacuuming had to be performed.

2 Implementation

The programming language used for the implementation of the project is 100% Python.

2.1 Project Dependencies

The project dependencies are as follows:

- '**rospy**' - Python client library for ROS. It enables for the quick prototyping and testing of algorithms within ROS;
- '**pathlib**' - provides classes representing filesystem paths with semantics appropriate for different operating systems [5];
- '**yaml**' - YAML parser and emitter library [7];
- '**math**' - provides access to various mathematical functions [4];
- '**tf**' - allows keeping track of several coordinate frames over time. It lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time via a tree structured buffer [14];
- '**queue**' - used in threaded programming. Implements multi-producer and consumer queues (FIFO and LIFO) [6];
- '**nav_msgs**' - defines ROS navigation stack interaction messages [11];

- '**std_msgs**' - standard ROS message package including common message types for primitive message types and data constructs such as multi-arrays [13];
- '**geometry_msgs**' - provides common geometric primitive (points, vectors and poses) messages [10];
- '**numpy**' - library used for working with array data structures (since Python lists are slow) [9];
- '**sensor_msgs**' - includes commonly used sensor messages. Such sensors are cameras and laser scan rangefinders [12];
- '**visualization_msgs**' - the main message of this set is the 'Marker' message utilized in sending visualization 'markers' (boxes, arrows etc.) to high level libraries such as rviz.
- '**copy**' - provides generic deep and shallow copy (with and without references to original data respectively) operations [3];
- '**sklearn.svm**' - includes Support Vector Machine algorithms (OneClassSVM for Unsupervised Outlier Detection) [8].

2.2 ROS and RViz Visualization

This task required displaying the robot's true and believed path as it moves around the map, as well as its true and believed location and orientation. In order to achieve this several Python code files were created which will be described below.

- '**path.py**' - implements the 'Path' class employed in creating robot trajectory instances of in different frames of reference. During object initialization a line and arrow markers are created representing the robot trajectory and the robot orientation respectively. Also, two publishers are initialized along with a subscriber which utilizes the '*callback(self, data, args)*' method. This function updates the trajectory and orientation markers with the odometric data received from the robot and publishes them.
- '**odom_path.py**' - initializes the odometry path node via the '*rospy.init_node()*' method and creates a 'Path' instance to reflect the 'believed' odometric (yellow) trajectory of the robot. The line marker is published as '/believed_path' while the orientation as '/believed_orient' to the '/odom' frame of reference in RViz.
- '**true_path.py**' - initializes the true path node via the '*rospy.init_node()*' method and creates a 'Path' instance to reflect the 'true' odometric (cyan) trajectory of the robot. The line marker is published as '/true_path' while the orientation as '/true_orient' to the '/base_pose_ground_truth' frame of reference in RViz.

2.3 Path Planning

For this task reading the instructions parameter which contains a list of (x,y) coordinates from a configuration file (YAML) was required. Then, these locations had to be drawn on the map (using RViz). Also, a path planning algorithm implementation was necessary to determine the 'efficient' path that the robot should drive along to get to these points. The developed files are as follows:

- '**pathfinder.py**' - represents an A* path planner implementation. This module includes:
  - '*a_star(init_coord, goal, map_cells)*' - the main function of the module representing the A* search algorithm. The first two arguments of the function are the start and goal (x, y) location coordinates of the path to be created. The last argument is the map passed as 'nav_msgs/OccupancyGrid' message so that it can be operated on. This algorithm is guaranteed to find the true shortest path since the adopted Manhattan distance heuristic is admissible (i.e. its estimated distance is always less than the actual distance) (astar wiki);
  - '*adapted_tsp(init_coord, g_queue, route)*' - develops the Adapted Travelling Salesman Problem. That is given a set of goal points and the initial robot coordinates it returns the shortest possible route that visits ever goal exactly once;
  - '*get_nbrs(curr, parent, map_cells)*' - returns a list of safe path point locations where the robot can drive to. The three arguments are the current and previous locations of the robot, and the 'nav_msgs/OccupancyGrid' message of the path planning map respectively;
  - '*is_safe_loc(curr_x, curr_y, parent, map_cells, nbrs, d=False, safe_dist=.12)* - checks if a neighbour location to the given robot position will allow for the movement and rotation of the robot without crashing to an obstacle. Returns a list of safe locations and variable 'd' which indicates if a diagonal cell movement is possible;
  - '*calc_heuristic(curr, g)* - calculates and returns the Manhattan distance heuristic (admissible thus 'efficient'). Arguments: 'curr' - the current location coordinates; 'g' - the goal location coordinates;
  - '*is_free(curr, map_cells)* - returns 'True' if the given 'curr' coordinate location is free from obstacles and 'False' otherwise.
- '**parse_config.py**' - YAML file parser, particularly 'config.yaml' to get the goal instructions and all the necessary ROS parameters. Also, visualizes the goal points on the map in RViz.
  - '*wait_for_time()*' - waits for the simulated time to begin via 'rospy.Time()' method;
  - '*show_goals_in_rviz(instructions, publisher)*' - visualizes goal markers in RViz by creating a 'Marker' message instance of type 'cube' and publishes them as 'goal_markers' to the 'odom' frame of reference in RViz;
  - '*main()*' - initializes the parser node and runs it.

Note that the path to the next goal is calculated only after reaching the current goal. Therefore, the whole route taken by the robot is not constructed at the start but each goal path is concatenated with the previous dynamically. Moreover, since there is a function which checks the occupied cells of the map it is ensured that the robot won't strike obstacles on the initial map (the PNG file) when following the path. For a detailed description of the above-mentioned functions and method please refer to the project module comments.

2.4 Driving and Vacuuming

For the purpose the path created in the previous task had to be followed having the robot stop at the last goal location point. Each goal was placed in separate room and vacuuming had to take place in each of these rooms. I only managed to clean approximately 50% of each room via spirally rotating the robot with vacuuming on. Moreover, the room floor to be vacuumed was drawn in different color while driving towards it. The vacuumed area of the room appears in blue in RViz through the use of 'Marker' messages. The solution comprise of the following modules:

- '**navigator.py**' - this is the robot navigator module where the 'Navigator' class is developed. It consists of the following methods:
  - '*laser_callback(self, data)*' - processes the robot laser scan data 'data'. It is utilized when the VC is in vacuuming mode to detect obstacles while spirally cleaning a room.
  - '*update_pose(self, odom)*' - callback method which updates the robot pose ((x, y) robot coordinates) with the received odometry data ('odom' argument);
  - '*update_pose(self, odom_pose*' - updates the robot pose with the odometry data 'odom_pose' and visualizes the vacuumed room area in RViz via markers;
  - '*vc_freeze(self)*' - stops the VC robot by publishing angular and linear velocity messages set to 0;
  - '*start_vacuuming(self, tsp_goal)*' - tells the robot to start spirally vacuuming the room. When an obstacle is detected the robot stops and drives to the goal point 'tsp_goal' and calculates the path to the next goal.
  - '*to_goal(self, g)*' - moves the VC robot to the given goal coordinate location 'g'
- '**cleaner.py**' - implements the VCleaner class employed in driving the vacuum cleaner to the specified goals while spirally vacuuming the target rooms where the goals are placed. This module initializes the 'cleaner' node, the robot pose, constructs the grid map, creates a robot navigator instance, assigns the necessary parameters and launches the robot driving and vacuuming processes.
  - '*prioritize_goals(self)*' - prioritizes the target goals by putting the closes goal first by using the 'PriorityQueue()' class;

- '*update_pose(self, odom)*' - callback method which updates the robot pose ((x, y) robot coordinates) with the received odometry data ('odom' argument);
- '*needs_charging*' - incomplete method (needs A* path planning implementation) which checks if the vacuum cleaner needs to be charged. If so drives to the closest charging point;
- '*run_vc(self)*' - main method which runs tha vacuum cleaner (VC) instance object. Drives the VC from goal to goal by following the constructed A* path. Also, draws the room floor and path markers, and handles the vacuuming done in each room;

The implementations were also tested and performed flawlessly with the odometric noise enabled in the 'world.world' file provided. For a detailed description of the above-mentioned functions and method please refer to the project module comments.

2.5 Localization

Robot Localization is the process of determining, where robot is located with respect to it's environment. Localization can be accomplished using different algorithms. In our case, the algorithm used in attempt to achieve the robot localization task was the particle filter (PF) algorithm. It involves a combination of Monte Carlo methodology, Bayesian Inference, and sensor processing. The PF starts with initializing normally distributed particle cloud around the robot's position. On receiving odometry updates, the particles are moved together with a small amount of randomness (noise) applied. On receiving a laser scan, the particle filter determines the likelihood of each particle to represent robot's true position. After the laser scan update, a best guess is determined by the particle filter for the vacuum cleaner position. Finally, the particle filter resamples the particles and the cycle starts again. The solution attempt implementation modules are provide below:

- '**pfl.py**' - this is the particle filter localization implementation module. It implements the 'Particle' and 'ParticleFilterLoc' classes. The former is employed in initializing particle instances representing possible VC robot poses. Its methods include:
  - '*conv_to_pose(self)*' - converts a particle to a 'geometry_msgs/Pose' message and returns it;
  - '*init_on_map(self, grid_map)*' - initializes a particle instance at some random location with a random orientation inside the map boundaries.

  The latter is the particle filter localization class which manages the following methods to carry out the localization:
  - '*rob_pos_update(self)*' - updates the estimated robot pose given the update particle weights (assigns the highest weight particle pose to the robot pose);
  - '*odom_ptcl_update(self)*' - initializes a particle instance at some random location with a random orientation inside the map boundaries.

  - '*get_inl_outl(self)*' - uses Scikit Learn OneClassSVM for unsupervised outlier detection. That is to choose which particles to 'ommit' and which to keep when resampling. Returns inlier and outlier lists;
  - '*resample(self)*' - resamples the particles according to the new particle weights. These determine the probability of each particle being selected in the resampling;
  - '*ls_pt_prob(dist)*' - static method which returns the probability of the laser to return the point at a given distance from the obstacle;
  - '*ls_ptcl_update(self, ls_data)*' - updates the particle weights based on the received laser scan data;
  - '*get_rand_sample(choices, probs, size)*' - static method which returns a random sample of the given size from the set of choices with the specified probability;
  - '*init_pose_update(self, msg)*' - reinitializes the particle cloud based on a RViz pose estimate;
  - '*init_ptcls(self, triple=None)*' - initializes the particle cloud then normalizes it and updates the robot pose;
  - '*norm_ptcl_weights(self)*' - normalizes each particle weight with the total particle weight;
  - '*pub_ptcls(self, msg)*' - visualizes particle poses by publishing a 'PoseArray' so that it can be seen in RViz;
  - '*laser_cb(self, laser_data)*' - processes the laser scan 'ls_data' data. Waits if initialization is not complete or transformation are not currently possible. Also, initializes the particle cloud if needed, updates particles and robot pose, resamples the particle cloud and publishes it.
  - '*base_odom_update(self, data)*' - update the odometry and map frame offset based on the localizer data;
  - '*bc_trfm(self)*' - broadcasts the odometry to map frame translation and rotation transformation. The localization was tested using the 'teleop' ROS node.

- '**obstacle_map.py**' - implements the 'ObstacleMap' class. Given an input grid map the class is used to generate obstacle map instances employed in 'extracting' distance to the closes obstacle for any map coordinate.

  - '*get_dist_to_obstacle(self, x, y)*' - return the closes obstacle distance to the specified (x, y) coordinate in the map.

2.6 Helper Modules

The following modules were implemented for convenience purposes:

- '**help_funcs.py**' - implements a range of functions used in the other modules. That is to achieve code flexibility and avoid unnecessary repetitions thus saving computational overheads.

  - '*eucledian_dist(pos, goal)*' - returns the straigh line distance between two points;
  - '*trans_rot_to_pose(transl, rot)*' - converts the given translation and rotation arguments to a 'geometry_msgs/Pose' message and returns it;
  - '*invert_pose_tf(pose)*' - inverts a pose transform and return the resultant translation and rotation tuple;
  - '*pose_to_triple(pose)*' - converts the given pose argument to a (x, y, yaw) triple and returns it;
  - '*norm_angle(angl)*' - return the angle mapped to the [-pi, pi] range;
  - '*get_rot_mat(theta)*' - return a 2x2 rotation matrix;
  - '*angle_diff(a, b)*' - returns the difference between angles 'a' and 'b'

- '**marker.py**' - 'Markers' class module used for creating the goal path and vacuumed room area line.

3 Design Decisions

3.1 Why A* Algorithm

A* is an advanced BFS algorithm that searches for the shorter paths first rather than the longer ones provided a decent heuristic such as Manhattan distance. A* is optimal meaning that it will surely find the least cost path from the source to the destination. It is also complete since it inevitably finds all the available paths from the source to the goal. Even though it is rather slow and space-demanding, it is one of the best path planning algorithms available [1].

3.2 Why Particle Filter

Particle filters have a number of advantages which have led to their popularity in current robotics. First of all, the algorithm is relatively easy to implement. Then, since the particles approximate the future belief, it gives a quite precise answer to any kind of expectation posed. Moreover, the particle filter algorithm can be utilized for any designed observation or motion model. Particle filters scale well and are truly parallelizable. Also, they work for high dimensional systems as they are independent of the size of the system [2].

4 Alternatives

4.1 Speed Up Localization and 'ObstacleMap' data structure

The particle filter needs some tuning since it behaves in a really slow manner. On the other hand, to make the obstacle map look up process faster we could maybe generalize the map points to lines.

4.2 Stationary localization

The implemented particle filter waits for a vacuum cleaner movement before resampling and updating particle positions. Therefore, updating particles while stationary could be advantageous.

4.3 Driving to Charger Points

Since a greater amount of time was spent on implementing localization, the driving to charger points around the map was left as a future work. So far only a callback method is implemented which gets the straight line distance to the charging point and follows that line. However, A* path finding should be also utilized to determine the closest safe path to the charging point.

5 How to Run

You should ensure that you have the ROS map server installed. Then, you should unzip the project files (nodes) into your catkin workspace and enter 'source devel/setup.bash' to the terminal from your workspace root to make sure that ROS can find the nodes. After that running the launch file would suffice since every project node is included in it.

To run the launch file while taking battery/power constraints into account run the launch file with the additional argument battery:=1. For example 'roslaunch assessment assessment.launch battery:=1'. To draw the locations where vacuuming has taken place, add the argument 'draw_vacuum:=1'.

6 Note

Apologies for the long report and the amount of bullet points. I really tried to make the report as neat and concise as possible and to implement as much as I could while spending a vast amount of my 'precious' time on this vacuum cleaner. I would really appreciate if you acknowledged this achievement of mine which might seem small to you but it is incredible one to me. As one would say: 'Not much but it's honest work...' Stay safe!

References

[1] Akash. A* algorithm — introduction to the a* search algorithm. https://www.edureka.co/blog/a-search-algorithm/. Accessed: 08 Jan, 2021.

[2] Drew Bagnell. Good, bad, and ugly of particle filters. https://www.cs.cmu.edu/~16831-f14/notes/F14/16831_lecture05_gseyfarth_zbatts.pdf. Accessed: 08 Jan, 2021.

[3] docs.python.org. copy – shallow and deep copy operations. https://docs.python.org/3/library/copy.html. Accessed: 08 Jan, 2021.

[4] docs.python.org. math – mathematical functions. https://docs.python.org/3/library/math.html. Accessed: 08 Jan, 2021.

[5] docs.python.org. pathlib – object-oriented filesystem paths. https://docs.python.org/3/library/pathlib.html. Accessed: 08 Jan, 2021.

[6] docs.python.org. queue – a synchronized queue class. https://docs.python.org/3/library/queue.html. Accessed: 08 Jan, 2021.

[7] pyyaml.org. Libyaml. https://pyyaml.org/wiki/LibYAML. Accessed: 08 Jan, 2021.

[8] scikit learn.org. scikit-learn 0.24.0 documentation. https://scikit-learn.org/stable/modules/classes.html#module-sklearn.svm. Accessed: 08 Jan, 2021.

[9] w3schools.com. Introduction to numpy. https://www.w3schools.com/python/numpy_intro.asp. Accessed: 08 Jan, 2021.

[10] wiki.ros.org. geometry_msgs - ros wiki. http://wiki.ros.org/geometry_msgs. Accessed: 08 Jan, 2021.

[11] wiki.ros.org. nav_msgs - ros wiki. http://wiki.ros.org/nav_msgs. Accessed: 08 Jan, 2021.

[12] wiki.ros.org. sensor_msgs - ros wiki. http://wiki.ros.org/sensor_msgs. Accessed: 08 Jan, 2021.

[13] wiki.ros.org. std_msgs - ros wiki. http://wiki.ros.org/std_msgs. Accessed: 08 Jan, 2021.

[14] wiki.ros.org. tf - ros wiki. http://wiki.ros.org/tf. Accessed: 08 Jan, 2021.