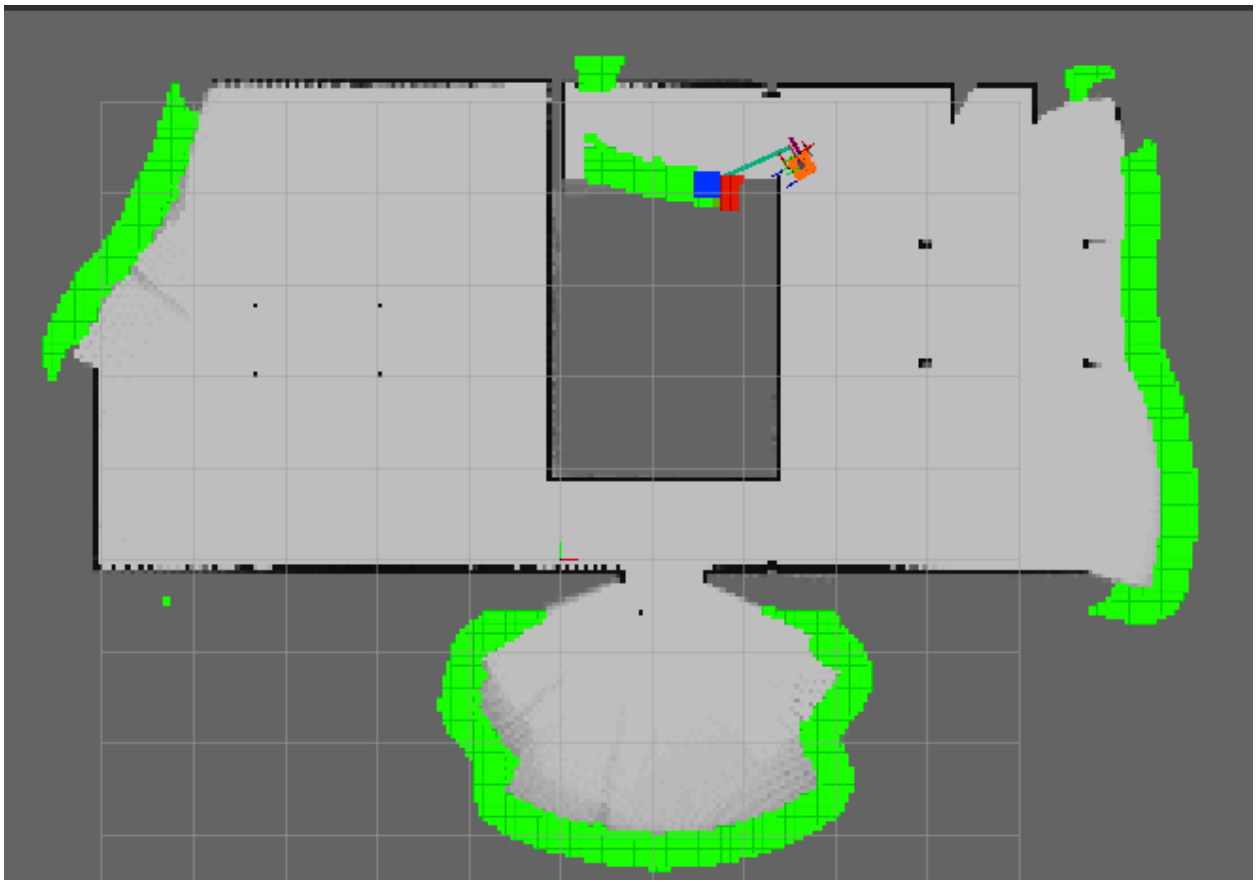
Autonomously Exploring Robot

Baaqer Farhat & Andrey Korolev



Project Summary:

This project aimed to explore some of the more advanced ideas concerning Roomba-type robots. In particular, we focused on map-based odometry correction accompanied by autonomous exploration. The implementation of the two mentioned features allows for a very comprehensive understanding of the surrounding environment even with noisy data – a necessity for autonomous robots.

The robot used was a turtlebot simulated in Rviz and Gazebo. The program was thoroughly tested in a household map environment, which was mapped to a 240x360 occupancy grid.

Initially, odometry correction used the Kabsch algorithm to align scan points with the map, but it overcorrected due to sensitivity to noise, causing unstable position adjustments.To improve stability, a least-squares method with KDTree-based matching was adopted. Gaussian noise is added to sensor readings, and corrections are gradually integrated into a map-to-drift transform, ensuring smoother real-time alignment.

The autonomous exploration was executed using frontier detection and a heuristic to select the next target to explore. Once a target is chosen, the robot moves down a sequence of path segments using a simple PD control loop. This approach explores the map relatively well given a properly tuned heuristic, however, it lacks sophistication in terms of recognizing that the target it is moving towards has already been sufficiently explored, which leads to inefficiencies.

In the end, the corrective localization and autonomous exploration worked very well together, resulting in a relatively accurate map and allowing the robot to explore without deviating significantly from the desired paths.

- **Project Goals:** Briefly describe the objective of the project.
- **Challenges:** Explain the problem statement and key difficulties encountered.
- **Key Ideas:** Outline the main concepts or innovations pursued.

Approach:

1. **MAP BASED Correction (noisy laser vs ideal map):**
   1.1 Coordinate Frame setup and Transform Handling:

Our localization approach begins by clearly defining the coordinate frames to isolate and correct odometry drift. The system is structured with the map frame as the global "root" frame—where RViz displays all information—followed by a drift frame, then the odom frame (which represents the robot's initial position as per its odometry), and finally the base frame attached to the robot (with the lidar mounted relative to it). In this setup, the odometry provides base w.r.t. odom while our "noisy localization" introduces errors via an artificial drift (odom w.r.t. drift).

Our algorithm computes the necessary correction (dx, dy, dθ) by transforming noisy scan points into the drift frame using the latest transform available (via Time()) instead of the scan's timestamp, which circumvents extrapolation issues. Visual markers are published in RViz to overlay the correspondence between the scan points and the map obstacles, aiding in real-time debugging and verification.

1.2 Noisy localization Correction:

To counteract the simulated noise, we extract scan points from the lidar and use a KDTree-based nearest neighbor search to match these points with map obstacles. The error is minimized via a linearized least-squares formulation that computes correction values for translation and rotation. Initially, we experimented with the Kabsch algorithm, but it often yielded overly high correction values. Switching to our current least-squares method provided more stable and realistic updates. Although our system correctly computes the inverse corrections, the corrected scan points do not consistently snap onto the map in real time, especially when noise increases beyond a threshold (e.g., 10% noise versus 5%).

2. **Autonomous Exploration**

Autonomous exploration was not explicitly discussed in the class, however we were nonetheless able to incorporate concepts such as trees, heuristics, and modified path planning in implementing this feature.

2.1 Frontier & Goal Selection

To efficiently explore the map, we needed a method for identifying unexplored areas. Given our 240×360 resolution, we opted for an intuitive, tunable method:

1. Identify all unknown cells surrounded by a sufficient number of clear cells and no blocked cells.
2. Rank these frontier cells based on the density of frontier neighbors, penalizing distance and previously attempted nodes.

This ensures that the robot prioritizes regions that maximize exploration value, are accessible, and do not force unnecessarily long movements. However, this approach struggles in enclosed spaces, as frontier nodes near obstacles may never be selected due to their limited connectivity. This could be mitigated by adjusting neighbor thresholds or implementing a more robust heuristic for goal selection.

2.2 Path Finding and Exploration

Once a goal is chosen, the robot must efficiently navigate to it. We use a modified RRT that prioritizes exploration over optimality—since discovering new regions is more important than the shortest path. RRT accelerates path-finding by biasing 30% of node selections toward the goal, significantly improving performance in open spaces.

After generating a path, the robot follows it completely to maximize exploration. However, this introduces inefficiencies: even if a goal region is already explored mid-travel, the robot still completes its planned path. A more adaptive approach would re-evaluate the goal in real-time, though this would increase computational overhead.

2.3 Path Post-Processing & Robot Control

To smooth the RRT path, we apply a simple two-pointer post-processing step that removes redundant segments, typically reducing paths to 1-6 segments. While curved paths could improve efficiency, we prioritized simplicity by sticking to straight-line navigation.

For movement control, the robot uses two PD controllers for linear and angular velocity. While this provides sufficient smoothness, tuning parameters for effective turning remained a challenge, likely due to publishing delays in Rviz.

Technical Details:

1. General Features

1.1 Map Generation and Lidar Scans

- ScanCB(): Invoked when a new laser scan message arrives. Before any correction is computed it checks the robot's angular velocity vs a **TURN_THRESHOLHD**. If the robot is rotating (> 0.1 rad/s) the function skips the update to avoid unstable correction during rotations. Also, rather than using lasers timestamp, ScanCB() fetches the latest transform from **drift** to the laser frame (**base_scan**) and it then adds Gauusian noise (**NOISE_STD_X, NOISE_STD_Y , NOISE_STD_Theta**) in an attempt to simulate real world sensor inaccuracies.
    - Point Conversion and Validity Check:
        - Each valid range reading that's within **range_min** and **range_max** is converted into cartesian coordinates relative to **drift** frame. Points that fail this check are discarded to reduce outliers.
    - Broadcasting and Verification:
        - ScanCB():composes the incremental correction with the current map → drift transform and broadcasts this new transform. A simple verification function (called verify_correction) logs whether a "significant" correction was applied.

1.2 Map Handling and Processing

- The script processes the occupancy grid from /map and reshapes it into a 2D NumPy array (self.map_array). These values range from 0 to 100, 0 signifying full confidence of an empty cell, and 100 signifying full confidence of an occupied cell.
- It sets thresholds (OBSTACLE_THRESH=80, CLEAR_THRESH=30) to classify free and occupied spaces. The given thresholds displayed reasonable performance in classifying ambiguous cells (in between the thresholds), which is desired in order to not miss any potential unexplored spaces.
- scale_coordinates(): Converts real-world coordinates to grid indices.
- unscale_coordinates(): Converts grid indices back to real-world coordinates.

2. MAP BASED Correction (Noisy lidar correction on existing map)

Our approach relies on aligning the noisy laser scan data to an ideal occupancy grid map by computing a small corrective transform d=[Δx, Δy, Δθ]^T that minimizes the alignment error. Given a set of laser scan points Q (transformed into the drift frame) and their corresponding nearest obstacle points P (extracted from the map), we compute the following averages:

$$R_x = \frac{1}{N}\sum_{i=1}^{N} Q_{i,x}, \quad R_y = \frac{1}{N}\sum_{i=1}^{N} Q_{i,y}, \quad P_x = \frac{1}{N}\sum_{i=1}^{N} P_{i,x}, \quad P_y = \frac{1}{N}\sum_{i=1}^{N} Q_{i,y}$$

We then define:

$$RR = \frac{1}{N}\sum_{i=1}^{N} (Q_{i,x}^2 + Q_{i,y}^2), \; RP = \frac{1}{N}\sum_{i=1}^{N} (Q_{i,x}P_{i,y} - Q_{i,y}P_{i,x})$$

And set the denominator as:

$$denom = RR - (R_x^2 + R_y^2)$$

The linearized least-square update is then computed via:

$$\Delta\theta = \frac{RP - (R_x P_y - R_y P_x)}{denom}$$

$$\Delta x = (P_x - R_x) + R_y \Delta\theta$$

$$\Delta y = (P_y - R_y) - R_x \Delta\theta$$

Finally, to obtain the correction from the map to the drift frame, we invert the signs:

$$\Delta x_{final} = -\Delta x, \qquad \Delta y_{final} = -\Delta y, \qquad \Delta\theta_{final} = -\Delta\theta$$

The key function, computeAlignmentLS(), implements the above model. It first retrieves the latest transform between the laser scan frame and the drift frame—avoiding timestamp extrapolation issues by using the current time. The function converts the laser scan into 2D points Q in the drift frame, then finds the nearest neighbor P for each point using a KD-tree built from the occupancy grid (converted from pixel to Cartesian coordinates).

After computing the averages and sums for Q and P, the function derives the correction update (Δx, Δy, Δθ) using the equations above. The update is then clamped to a maximum norm (MAX_UPDATE_NORM) to avoid abrupt changes. In the scanCB() function, this update is integrated gradually into the global transform map→drift by composing it with the current

drift values. The updated transform is broadcast via TF so that the "corrected" laser data (and thus the drift frame) aligns with the map. Key state variables include the current drift correction parameters (dx_map, dy_map, dt_map), the scan points Q, and the matched map points P. This setup ensures that the noisy scan data is iteratively "snapped" to the ideal map.

2.1: **computeAlignmentLS():** This function takes the latest laser scan that's already converted into the "drift" frame) and computes how well the noisy points align with the known map obstacles

- **Noise injection:** as mentioned in section 1.1
- **Nearest neighbor matching:** Each laser point is paired with its nearest obstacle point (found via a KD-tree of occupied cells). This yields a set of correspondences between scan data and map points.
- **Error Computation:** the mean euclidean distance between each laser point and its corresponding obstacle point is computed. If the mean distance is too large (above **MAX_MEAN_ERROR**), the function skips the update to avoid disruptive corrections
- **Least Square alignment:** calculates the small transformation delta x delta y delta theta needed to reduce the discrepancy between scan data and the map, it gives us how the "drift" frame should be adjusted to the "map" frame

2.2: **broadcastmapCorrection()**: Once the (Δx,Δy,Δθ) is calculated, the code updates and rebroadcasts the map→drift transform. This is how the system "moves" the robot's perceived frame relative to the global map. By continually refining map→drift, we gradually align the noisy laser data with the ideal occupancy grid. This transform is published to TF, allowing other nodes (and RViz) to see the corrected position.

3. Autonomous Exploration

3.1 Frontier Detection

- The robot detects unexplored areas by identifying grid cells with ambiguous values (between the clear and blocked thresholds) and ensures that it is mostly clear within a certain radius.
- The function get_frontier_idxs():
    - unexplored = map cells between the clear and blocked thresholds

    - clear_neighbor_count = array of # of clear cells surrounding each unexplored cell within the FRONTIER_RADIUS
    - blocked_neighbor_count = array of # of blocked cells surrounding each unexplored cell within the FRONTIER_RADIUS
    - frontier_indices = indices of cells s.t (clear_neighbor_count > CLEAR_NEIGHBOR_FRONTIER_THRESH) & (blocked_neighbor_count == 0)
    - Return frontier_indices

- Tunable constants:

    - FRONTIER_RADIUS = 8: neighbors are checked within an 8-cell radius. This proved sufficient when checking for surrounding cells; decreasing this number meant not having enough information around the unexplored cell and increasing it meant having a lot of potentially irrelevant information.
    - CLEAR_NEIGHBOR_FRONTIER_THRESH = 5: we want at least 5 of the surrounding cells to be clear. This constrains frontier nodes to not be far from explored areas and remain actual frontier nodes. Changing this number would essentially change the thickness of the frontier, and we found that 5 would provide a reasonable thickness for good exploration.

3.2 Goal Selection

- From the frontier cells, the robot selects the best one based on the following heuristic:

Cost = ((DIST WEIGHT) * (dist to cell) - frontier neighbors)) / (attempt weight)

- This heuristic prioritizes frontier cells that are surrounded by a lot more frontier cells; we want higher-density unexplored areas. Additionally, it punishes frontier cells that are too far away from the robot.

- Attempt weight is a weight assigned to each cell based on how frequently RRT failed to find a path to it and how frequently the robot had to abort path following due to a newly discovered obstacle. All weights are initialized at 1 and are decremented by 0.2 (within a surrounding radius) each time the aforementioned cases occur. This forces the robot to prioritize other cells when it fails to reach a certain cell several times.

- The function get_goal():

    - Calculate Euclidean distances between the robot to all candidate cells (from frontier list)

- Find the number of surrounding frontier neighbors for each frontier cell
- Calculate the cost for each frontier cell as defined above
- Select the frontier cell with the least cost

- **Tunable constants:**
  - HEUR_DISTANCE_WEIGHT = 5: since the distance to a cell has different units than the number of surrounding neighbors, the distance must be scaled. The number of surrounding cells can range from 5 - 50, so we scaled the distance by 5 to balance the distance and neighbor density. This results in prioritization of neighbor density until the robot is too far away.
  - FRONTIER_WEIGHT_DECREMENT = 0.2: the value by which the frontier weight is decremented. Thus, after 5 attempts, the weight of a cell is reduced to 0 (rounded to 0.01 to prevent division error).

3.3 Path Planning via RRT

- rrt(startnode, goalnode): Generates a path using an RRT-based approach:
  - Selects a random target point (biased 30% towards the goal).
  - Finds the nearest existing node in the tree.
  - Moves a fixed step towards the target.
  - Validates whether the new node is in free space and connects to the previous node.
  - Connects to the goal if the current node is within step distance.
  - Repeats until reaching the goal or exceeding step limits.

- Changes to Node class and RRT:
  - When checking if a node is in free space or connects to the next, use a hitbox with dimensions BOT_WIDTH_CLEARANCE and BOT_LENGTH_CLEARANCE. This ensures that the robot can actually fit in the desired cell with additional clearance.
  - Distance between nodes is kept as Euclidean since the robot can travel in all directions. In retrospect, some kind of turning distance could have also been added into consideration, much like in the mattress assignment.
  - The step from the current node to the next is calculated given the below equation. Calculating the step this way allows to scale the normalized step by some constant, which allows for easy tuning:

step = RRT_STEP_SCALE * (target_coords - curr_coords) / norm(target_coords - curr_coords)

- Tunable Constants
    - RRT_GOAL_SELECT_FRACTION = 0.3: since the map we use is not too cluttered, we chose to point the tree in the direction of the goal 30% of the time to expedite path-finding. Any lower and it would take considerably longer for the algorithm to find a path. Any higher and it would cause the algorithm to get stuck in obstacles and walls.
    - RRT_STEP_SCALE = 4: a step of 4 was chosen because it is big enough to allow RRT to find a path quickly enough while still ensuring a more optimal path (as opposed to massive steps that jump all around).
    - BOT_WIDTH_CLEARANCE = BOT_LENGTH_CLEARANCE = 10: while the actual width = height = 7, we want the robot to have some clearance when following the path, so the chosen value was 10 for the hitbox.
    - SMAX = 25000: the number of allowed steps. This value worked well for the given map size and prevented RRT from wandering for too long.
    - NMAX = 10000: the number of allowed nodes. This value also worked well for the given map size and was reached only when the goal was completely out of reach.

3.3 Movement and Collision Detection

- is_colliding(position, direction): Predicts collisions by checking a grid region ahead of the robot. Since the path generated by RRT isn't guaranteed to be clear, this is a necessary step to prevent collisions during the path-following phase. This works very similarly to the connects_to() method where we check if an extended hitbox (using COLLISION_CLEARANCE) is occupied by an obstacle a certain distance from where the robot is headed. This hitbox is checked using a step = COLLISION_STEP_SCALE * desired_direction away from the robot's current location. This allows the robot to stop before it collides with a newly discovered obstacle. If this function returns true during path-following, the sequence is aborted and a new goal-searching cycle begins.

- Robot controller: the robot controller uses two PD controllers to control the robot's linear and angular velocities. It loops through each segment of the path, calculating the desired direction and driving the robot forward when the angle is within a 0.1 rad tolerance of the desired angle. With every iteration, it calls is_colliding() and quits if the method

returns True, setting the velocities to 0 in order to stop. When finished with a path, the controller resets its values.

- Tunable Constants
    - COLLISION_STEP_SCALE = 9: this step scaling proved to be far enough way so that the robot would not collide but also short enough so that the robot wouldn't preemptively abort the path-following due to a distant obstacle that posed no threat.
    - COLLISION_CLEARANCE = 9: this value is slightly smaller than the width/length clearance used for RRT. This is because we want to provide a tiny bit of tolerance for the path provided by RRT, which should be clear within a width and length of 10 cells.
    - Kp_lin = 1: the proportional gain for the linear velocity
    - Kd_lin = 0.3: the derivative gain for the linear velocity
    - Kp_ang = 1: the proportional gain for the angular velocity
    - Kd_ang = 0.2: the derivative gain for the angular velocity
    - goal_tolerance = 0.05: goal tolerance (in meters)
    - angle_tolerance = 0.1 rad

3.5 Main Loop

- The loop() function handles the main execution cycle:
    - Waits for valid map and odometry data.
    - Selects a goal using frontier-based exploration, exits if no goal is found (finished exploring).
    - Computes a path using RRT.
    - If a path is found, post processes it and sends velocity commands to follow the path using RobotController.

Contributions and Lessons Learned

Localization

The key takeaways from our map-based correction method are that gradual, incremental updates enable more stable alignment of noisy laser scan data with a known occupancy grid, and that carefully clamping update magnitudes prevents abrupt, crazy transformations. Compared to methods like the Kabsch algorithm—which tended to yield overly large corrections sensitive to noise—our linearized least-squares approach, augmented by Gaussian noise modeling, provides robust performance even under moderate sensor uncertainty. Our performance analysis shows that while the algorithm reliably reduces drift when noise levels are low (e.g., 0.05 standard deviation), increased noise sensitivity at higher levels (e.g., 0.12) underscores the importance of tuning parameters like the update factor and maximum correction threshold. In addition, visual markers for scan-to-map correspondence offer immediate feedback, further supporting effective debugging and continuous performance evaluation.

Autonomous Navigation

The autonomous navigation that we implemented worked relatively well in finding densely unexplored regions near the robot. Frontier-based exploration is one of the most common methods when it comes to robot exploration because it provides the robot with a relatively contained set of points that it can then apply a heuristic to in order to continuously explore the map's boundaries. Our method focuses on intuition to provide easily tunable parameters and achieve effective exploration in different environments, which also causes computational inefficiencies and occasionally limited exploration in tight spaces. The algorithm we currently have could definitely use additional tuning for the house map to account for these drawbacks, in addition to more advanced calculations such as accounting turning distance into total distance between nodes and refining the frontier selection heuristic (calculating distance based on actual potential travel distance and not a distance that requires clipping through walls).

Conclusions

We are extremely satisfied with how well the localization correction and autonomous exploration correction. Future work could include completing the recommendations in the lessons learned section, in addition to adding a pinging feature between two robots in order to improve localization.