American University of Sharjah

College of Engineering

Department of Computer Science and Engineering

Spring 2024

CMP 49410 - Intelligent Autonomous Robotics

Project 3

Autonomous Can Detection and Mapping with RRT* and YOLO

Ahmad Alsaleh - @00093749

Ahmed Alabd Aljabar - @00092885

Omar Ibrahim - @00093225

Yousef Irshaid - @00093447

Submitted To: Dr. Michel Pasquier

Submission Date: 23rd May 2024

Installation

Six libraries were used to implement the project: OpenCV, NumPy, PyTorch, Matplotlib, Scikit-learn, YOLOv5. All libraries must be installed before running the simulation, which can be done using
`pip install -r requirements.txt`.

It is worth noting that if YOLO is not already installed, it will be downloaded and installed **automatically** when running Project 3's simulation. This may cause the simulation to take a while to start.

To end the simulation, **pressing 'Q' on the keyboard is required**. This will then save the final map in the current working directory as `map.png`.

Overview

This project implements an autonomous e-puck robot for the task of can detection and mapping using APF for reactive functionality, RRT* for path planning, YOLO and SIFT for object detection and recognition, Webots Display Device for real-time visualization of the robot's path and the detected objects, and finally DBSCAN clustering for map enhancement by removing inaccurately detected objects (i.e., noise). All implementation details are discussed thoroughly in this report.

Reactive Layer

For the reactive layer, we used the same AFP controller we used in projects 1 and 2. The APF defines two fields: an attractive field, and a repulsive field. The attractive field is applied to the next waypoint the robot is supposed to reach. It is implemented by subtracting the robot's coordinates (position vector) returned from the GPS from the position of the waypoint. This yields a vector that attracts the robot towards the waypoint. This vector is then scaled by a specified parameter, to control the force of the attraction. The repulsive field is calculated from the distance sensor values and the angle at which the sensors are placed around the robot. This vector is similarly scaled by a specified parameter to control the force of repulsion. The vectors are then added together to result in the target vector (where the robot should head to).

The speed of the robot is controlled by two components: the angle between the current robot orientation and the target vector, and the distance between the robot and the next target. An

angle of 90° and above means that the robot should make a sharp turn, and hence the speed is set to slow down as the angle increases so that the robot can make safer turns. On the other hand, if the angle is close to 0°, it means that the goal is ahead of the robot; thus, the speed is increased as the angle reaches 0°. Furthermore, when the robot is further away from the waypoint the speed of the robot increases due to the distance component. Both components are then combined using a weighted average:

```
raw_speed = 0.8 * raw_speed_angle_component + 0.2 * raw_speed_distance_component
```

Raw speed denotes the speed that is fed directly to both left and right motors, meaning it is what drives the robot forward. Afterwards, the left and right motors are added/subtracted another value proportionally controlled by the difference between the orientation of the robot and the target vector angle, which results in turning the robot left or right. The distance component was added to the raw speed because, after testing, we found that the robot gets stuck sometimes when it has to make a 180° turn and is beside an obstacle. The problem happens because the raw speed is 0 (meaning the robot doesn't move forward[1]) and it is avoiding an obstacle so it gets stuck in rotation movement in opposite directions. The distance component solves this problem by driving the robot forward slightly so that it eventually gets unstuck.

Deliberative Layer

Path Planning

The deliberative layer is similar to Project 2, but we don't have the map as a graph. We only store the vertices of the obstacles as the map representation. We also don’t construct a graph representation of the map we have, and so graph search path planners won’t work in this case. Therefore, we resort to RRT* for path planning as it only requires the obstacle map to work. The implementation of RRT* was taken from this excellent GitHub repository. The code assumes that all obstacles are circular, so we modified the code to support rectangular obstacles. The implementation of collision detection was also made for circular obstacles but was accordingly modified to suit our needs. The implementation also takes the robot radius into account while

[1] Note: even at raw speed of zero the robot rotates because the left and right motors are controlled using the target vector value.

checking for collision, so growing the map was not necessary because we programmatically grew it when checking for collision. A sample run of RRT* is shown below:

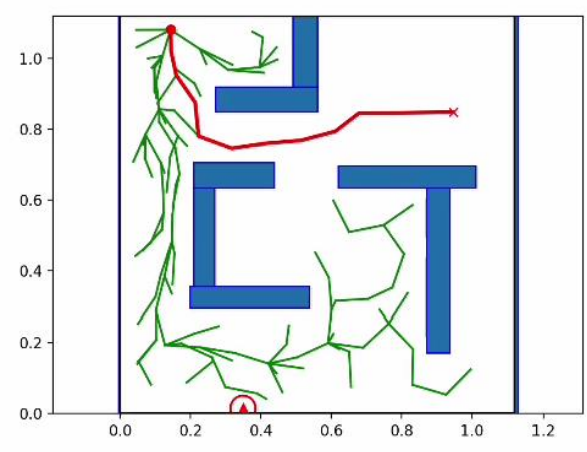

Figure 1: RRT* path planning demonstration

The red dot represents the start location and the cross represents the goal. The green branches represent the randomly generated sample and the red path is the final path from start to goal.

In order to simulate a wandering behavior, we generate a random goal for the robot to reach, and when it reaches we generate another. The problem that immediately arises is if the goal lies inside one of the obstacles, the path generation will keep trying to find a path but will never be able to. To solve this we check if the randomly sampled goal is inside an obstacle using the ray-casting algorithm:

```
def __is_inside_obstacle(
        self, point: Point, obstacle: Tuple[Point, Point, Point, Point]
    ) -> bool:
        """Checks if the point is inside the given obstacle."""
        count = 0
        for i in range(len(obstacle)):
            a, b = obstacle[i], obstacle[(i + 1) % len(obstacle)]

            if ((point.y < a.y) != (point.y < b.y)) and point.x < (b.x - a.x) * (
                point.y - a.y
            ) / (b.y - a.y) + a.x:
                count += 1

        return count % 2 == 1
```

The algorithm simply concludes that if a ray drawn horizontally from the point intersects with an odd number of segments, then the point is inside the polygon, otherwise it is outside as demonstrated below:

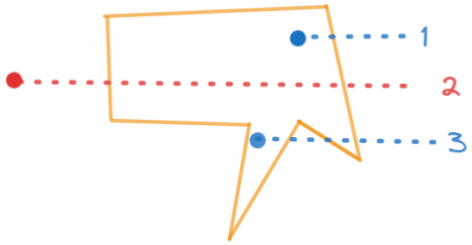

Figure 2: ray casting algorithm working principle

To ensure the correctness of the goal setting, we enlarge the obstacles before checking of they are inside:

```
def is_inside_obstacle(self, point: Point, robot_radius: float = 0) -> bool:
        """Checks if the point is inside any of the obstacles."""
        for obstacle in self.__obstacles:
            if self.__is_inside_obstacle(
                point, self.__enlarge_obstacle(obstacle, robot_radius)
            ):
                return True
        return False
@staticmethod
    def __enlarge_obstacle(
        obstacle: Tuple[Point, Point, Point, Point], robot_radius: float
    ) -> Tuple[Point, Point, Point, Point]:
        """Enlarges the given obstacle by the robot radius."""
        top_left, _, bottom_right, _ = obstacle

        enlarged_top_left = Point(top_left.x - robot_radius, top_left.y + robot_radius)
        enlarged_bottom_right = Point(
            bottom_right.x + robot_radius, bottom_right.y - robot_radius
        )

        enlarged_top_right = Point(enlarged_bottom_right.x, enlarged_top_left.y)
        enlarged_bottom_left = Point(enlarged_top_left.x, enlarged_bottom_right.y)

        return (
            enlarged_top_left,
            enlarged_top_right,
            enlarged_bottom_right,
            enlarged_bottom_left,
        )
```

Nevertheless, this all works well until an interesting case happens, which we encountered while testing. The way we enlarge the rectangle is by simply growing all the edges by the robot radius. However, this in fact is not true for an accurate configuration space. The actual representation of the rectangle in the configuration space is a rounded rectangle, the case is represented below:

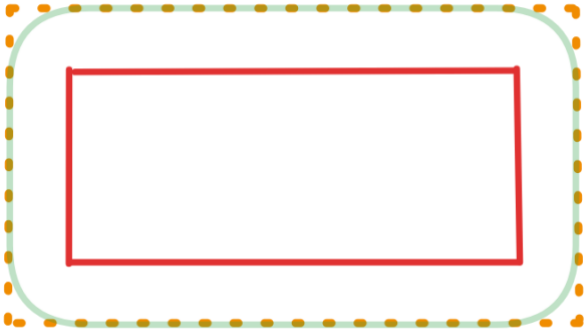

Figure 3: real configuration space vs assumed configuration space

Our robot can stop at a region between the green rounded rectangle (actual c-space) and the orange rectangle (assumed c-space). The problem here is if the robot stops at these locations, the root of the RRT* tree will be inside an "assumed" obstacle when in fact it is not. This will lead the RRT* algorithm to not connect any node to the root because it is detecting a collision, and the path planning will keep failing until the simulation is restarted. We have not solved this problem yet because it is a very rare edge case.

Object Recognition & Detection

To facilitate the recognition and detection of cans around the robot's environment, we experimented with and implemented two different state-of-the-art approaches for the task of object detection: a hand-crafted approach and a deep-learning approach. Despite the evident superiority of the deep learning approach in previous object detection tasks, the hand-crafted approach is still utilized in some applications when computation constraints are present. Therefore, we allow the user to pick the preferred object detection method before running the simulation by setting the global variable `OBJECT_DETECTION_ALGORITHM` to either YOLO (default) or SIFT. Additionally, if the user does not require any object detection whatsoever, it can be disabled through the global `ENABLE_OBJECT_DETECTION`, which is set to true by default.

Note that while both methods are implemented, YOLO is preferred as it is consistently more accurate than the hand-crafted method.

Hand-Crafted Approach

While many hand-crafted methods for object recognition and detection have been previously proposed, from simple histogram approaches to sophisticated algorithms, we decided to go for a technique that has been successful in the past, utilizing Scale Invariant Feature Transform (SIFT) as our feature detector and describer alongside the concept of homography matrices for image transformation. Our proposed method works by utilizing a reference picture, that is a clear picture of the can acquired a priori (Figure 4).

Figure 4: Reference Picture used for Image Matching

SIFT is then used to describe the reference picture and compare that to those obtained from the scene. Frame-skipping of 8 frames was utilized to ensure the algorithm does not affect the robot's speed. Thus, every 8 frames, the camera provides a scene image that is described using SIFT. The descriptors are then compared and matched using a brute-force matcher. If the descriptors show relevant matches, it suggests that the image from the scene potentially contains a can. To know the location of the can (i.e., coordinates), RANSAC is used to estimate the homography matrix. RANSAC, or Random Sample Consensus, is an iterative method that estimates the homography matrix (i.e., model) by testing the data against the fitted model until the estimated model is good enough. This method is robust to noise as it classifies the points as inliers or outliers, making it suitable for our application. If no inliers are found out of the matched key points, RANSAC concludes that the can was not detected (i.e., the model is rejected). If however, RANSAC classifies most points as inliers, it confirms that the can is in fact present and returns the homography matrix that allows the reference target image to be transformed into the scene image. This transformation is then applied to obtain the coordinates of

the can in the scene image. The process is illustrated visually in Figure 5 and Figure 6. Note that this visualization does not take place when running the algorithm. Finally, given the coordinates, the distance and angle from the camera are computed using the `compute_distance_and_angle(self, x_min, x_max, scene_image)` function, which is discussed later.

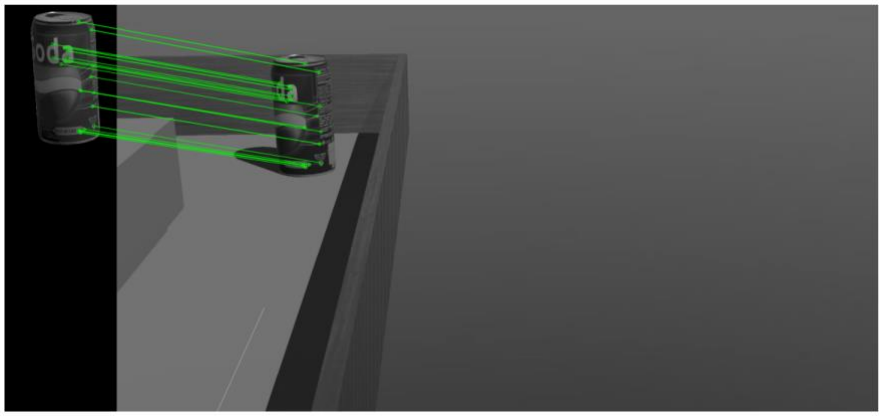

Figure 5: Image Matching using Brute Force Matcher on SIFT Descriptors

Figure 6: Coordinates of Detected Object Found using RANSAC for Homography

Deep-Learning Approach

The state-of-the-art model You Only Look Once (YOLO) was selected as our main object detector. In particular, YOLOv5 was utilized, which is pre-trained on the COCO dataset. The COCO dataset contains 80 classes including cups and bottles, which are very similar to a can. Since the model is adequate in detecting cans, as will be discussed, it was used as is with no finetuning (i.e., transfer learning). Four classes were identified to be correlated to a can (i.e., the prediction of the model when it sees a can) that are: cup, bottle, cell phone, and book. Once one

of these classes is detected by the model on the image from the scene, it is said that a can is detected. Similar to the hand-crafted approach, frame-skipping of 8 frames was done for memory constraints. However, this did not affect the detection of the model and can be seen as an enhancement to the system. Figure 7 shows an example of the bounding box returned by YOLO once a can is detected. The coordinates of the detected object, which are obtained directly from the bounding box, are then transformed to distance and angle using the `compute_distance_and_angle(self, x_min, x_max, scene_image)` function.

Figure 7: Example Bounding Box Returned by YOLO

Distance and Angle Computation

Compute_distance_and_angle function was implemented to compute the distance and angle of the target object in the scene image (i.e., polar coordinates) from the coordinates returned by YOLO or SIFT. The function takes in the x coordinates alongside the scene image (i.e., the image with the detected object). The function then computes the distance by utilizing the focal length of the camera, which is obtained using the field of view of the camera with the following formula:

```
        self.__focal_length = self.__target_image_gray.shape[1] / (
            2 * np.tan(self.__fov / 2)
```

The distance between the camera and the object is then calculated using the formula:

```
        distance = self.__focal_length * self.__target_size / target_size
```

Finally, the angle is then computed using the center of the image and target alongside the camera's field of view as seen below:

```
        angle = (target_center - image_center) / image_center * (self.__fov / 2)
```

The angle relative to the camera is represented in radians, with 0 implying the object is in the center of the image, a positive value indicating that the object is to the right of the image, and a negative value suggesting that the object is to the left of the image.

This is still not enough information to mark the object on the map, which is the objective of our object detection. For that, another function is created:

`draw_detected_objects_on_display(self, location: Tuple).`

This function takes the distance and angle as returned from the previous function. It then calculates the coordinates of which this object is to be placed by taking into account the robot's current position and angle. The x and y coordinates are calculated using the following formula which is a mere conversion from polar coordinates to Cartesian coordinates:

```
x = robot_x + distance_to_goal * np.cos(robot_angle)
y = robot_y + distance_to_goal * np.sin(robot_angle)
```

Finally, the obtained coordinates are mapped to the display's scale and then drawn on the display as seen below:

```
x, y = self.__map_to_display(x, y)
self.__detected_objects.append((x, y))
self.__draw_detected_objects((x, y))
```

Visualization

Two approaches were taken to visualize the path, the map itself by visualizing the final path chosen by RRT* and the display device which showcases the map alongside the chosen path by RRT*, and the locations of the detected cans.

Map

In the simulation environment, the RRT* path is visualized by drawing nodes and edges directly on the map. This is done once the RRT* algorithm finishes execution and finds the optimal path. A sample path is shown in the figure below:

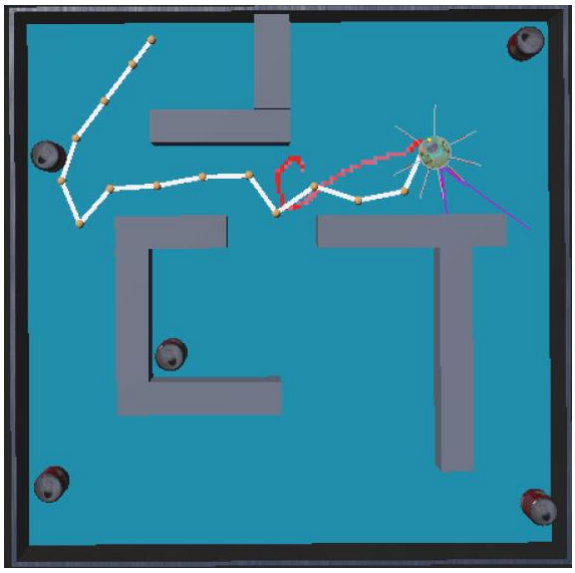

Figure 8: RRT* path visualization on the map

Once the goal is reached, the path is cleared and a new one is created to indicate the next trajectory for the robot.

Display

In addition to the direct visualization on the map, the display device was also used to combine all information in a compact form. First, the obstacles are drawn on the display as depicted below:

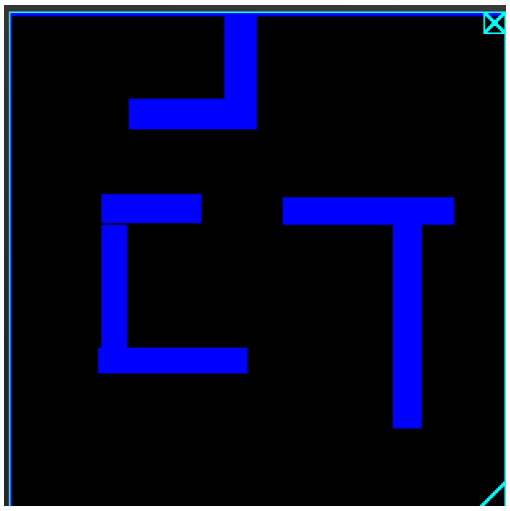

Figure 9: obstacles drawn on the display device

Additionally, the exact path taken by the robot is represented by a continuous red line as shown below:

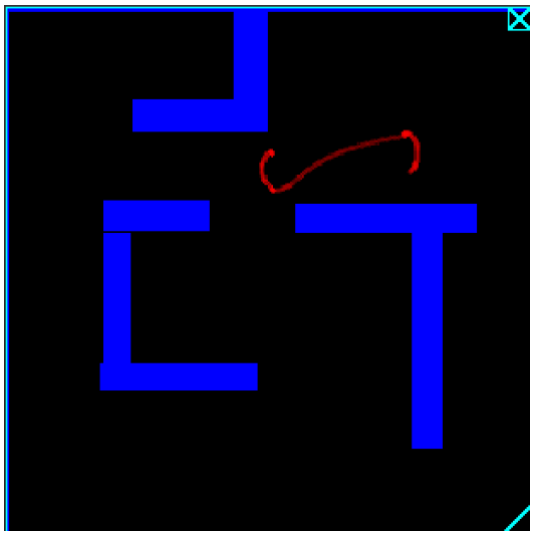

Figure 10: traversed path visualization on the display

Furthermore, the RRT* path is visualized on the map as shown below (in green):

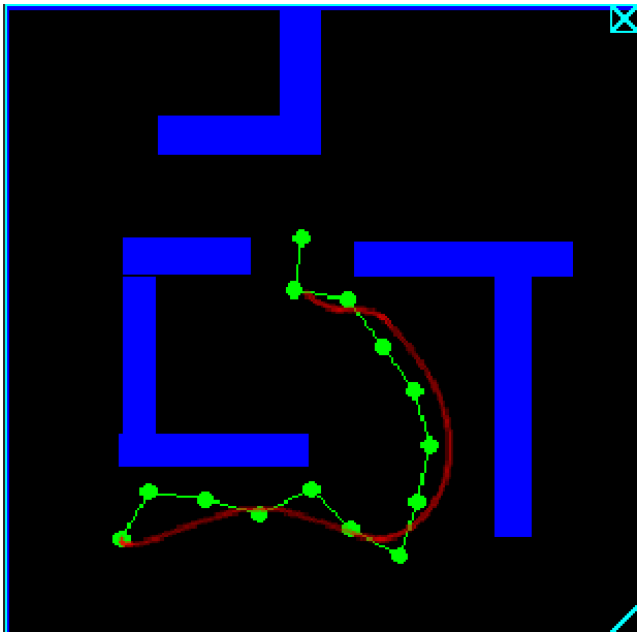

Figure 11: RRT* path visualization on the display

Whenever the robot detects a can using the object detection model, the approximate location is marked on the display using magenta circles. The higher the intensity the more probable that the location is indeed the correct can location.

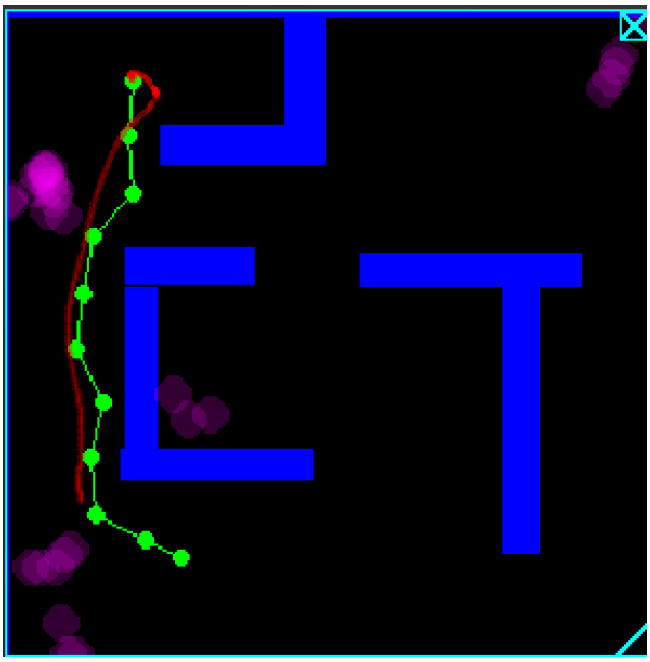

Figure 12: locations of cans drawn on the display as seen by the robot

Clustering

After the robot is done roaming for the objects and the simulation is ended (by pressing ‘Q’), clustering is performed on the approximate locations in order to produce one accurate location per object. To elaborate, all the detected locations are fed into the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) clustering algorithm with the minimum number of samples set to 2 and epsilon (i.e., neighborhood radius) set to 30. Following that, the centroids of the predicted clusters are extracted and used to represent the final locations of the objects. Figure 13 shows the final result of DBSCAN on the sample map.

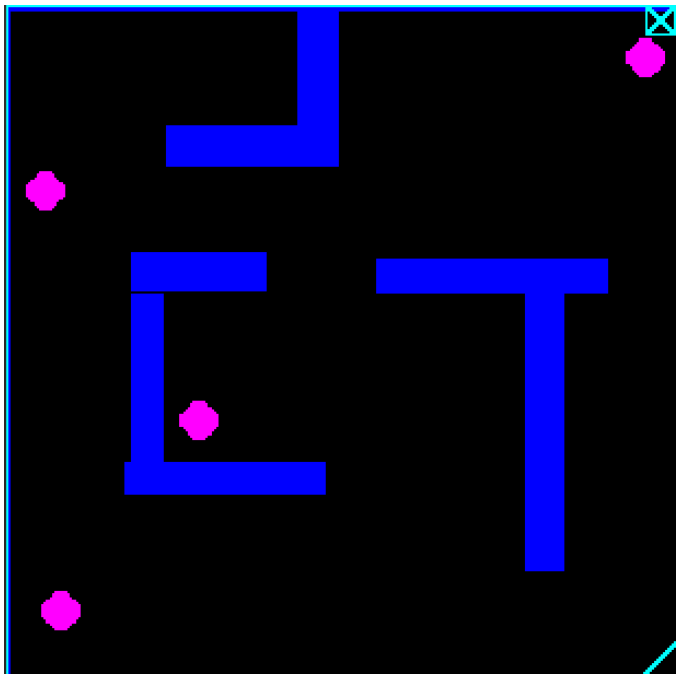

Figure 13: locations of the cans on the display after clustering