TP4 - AI FOR ROBOTICS

Master SETI

DO NASCIMENTO SANTOS Alaf

2023/2024

Contents

1 Introduction

As part of the AI for Robotics module, the main aim of this work is to study Laser telemetry and localisation using a differential-motion robot, Thymio model, simulated with the **Webots** software version 2021b and **Python** 3.9v.

Below are the objectives of the practical work:

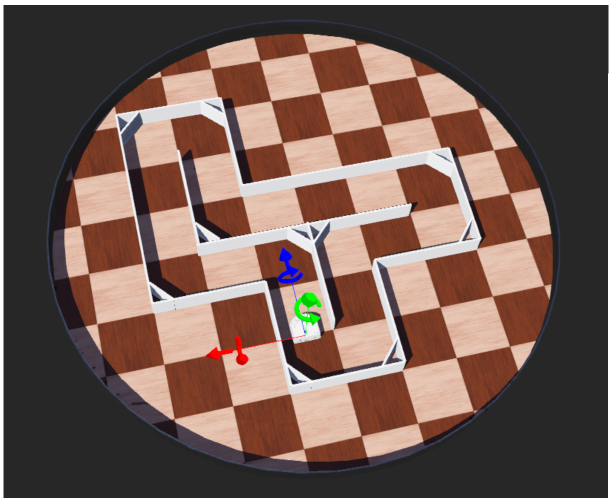
- Implementing a follow the gap - with bubble method
- Optimising the method as far as possible
- Testing the method by adding obstacles

Five Python files have been created for the project:

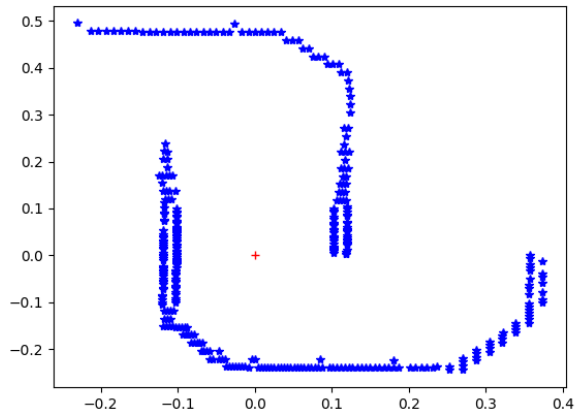
- **tp4.py**: This file serves as the main controller, orchestrating the system's behavior.
- the main controller interfaces with:
  - **flags_file.py**: It is used to enable or disable features such as debugging and keyboard control.
  - **graph_walls.py**: This file defines the `graphWalls` class, which is responsible for plotting functionalities using matplotlib.
  - **motors_controller.py**: Here, functions related to controlling the robot's motors are implemented. This file facilitates the movement and rotation of the robot.
  - **lidar_controller.py**: Contains functions related to the LIDAR sensor. These functions were either developed or adapted from the provided assignment and enable the robot to interface with and utilize LIDAR data.

2 Laser rangefinder - Follow the gap!

In this section, we want to explore the onboard laser telemeter embedded in the robot for a follow the gap reactive algorithm, without any odometry information. We start by resuming our previous program with the laser rangefinder, in which we plot the LIDAR data onto the animate Figure. As a result of this step, for instance, Figure 1a represents the actual position of the robot in the simulation space, while Figure 1b illustrates the acquired data, where the red cross represents the robot position and the blue marks are the walls around it (given by the LIDAR distances information).



(a) Robot position in the simulated environment.



(b) Measured LIDAR data.

Figure 1: Displaying LIDAR data in rectangular coordinates with no other treatment

Now we will concentrate on the laser data at the front of the robot, the idea is to delete the data at the rear of the robot and then display only the data at the front of it, as shown in Figure 2. Here the only thing necessary was to discard the points from the rear of the vehicle (points between $\frac{\pi}{2}$ and $\frac{3}{2}\pi$).

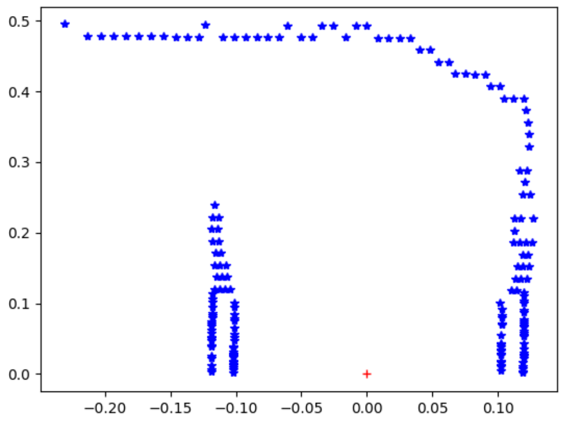


Figure 2: Initial laser rangefinder data acquisition of desired angles.

Using the previously designed algorithm, we want to identify the largest "Gap" in the laser data set. We need to iterate over the list of points and calculate the distances between consecutive points. The largest gap will correspond to the largest distance between any two consecutive points. The following function was used for this step:

```python
def find_largest_gap(lidar_data):
    # Initialize variables to store information about the largest gap
    largest_gap_distance = 0
    largest_gap_index = -1

    # Iterate through the points to find the largest gap
    for i in range(len(lidar_data) - 1):
        # Calculate distance between current point and next point
        dist = distance(lidar_data[i], lidar_data[i + 1])

        # Update information if the current gap is larger than the last
    one
        if dist > largest_gap_distance:
            largest_gap_distance = dist
            largest_gap_index = i

    rotation_angle = angle(lidar_data[largest_gap_index], lidar_data[
    largest_gap_index + 1])

    # Remove values equal to infinity
    if largest_gap_distance == np.inf:
        largest_gap_distance = 0

    if rotation_angle == np.inf:
        rotation_angle = 0

    if flags["debug"]:
        print("Largest gap distance:", largest_gap_distance)
        print("Index of the point where the gap starts:",
    largest_gap_index)
        print("Rotation angle:", rotation_angle)

    return largest_gap_distance, largest_gap_index, rotation_angle
```

Subsequently, we want to rotate the gap to position it facing the robot. The following function was used together with the previous code to find the rotation angle:

```python
# Function to calculate angle between two points
def angle(point1, point2):
    return np.arctan2(point2[1] - point1[1], point2[0] - point1[0])

rotation_angle = angle(lidar_data[largest_gap_index], lidar_data[
    largest_gap_index + 1])
```

Then we want to move the robot forward and then complete the search for a new Gap. Since this operation, of searching for a new target point (composed of rotation and translation) doesn't need to be done at each iteration, we introduce some flags for only doing it when needed (after finishing the current rotations and translations. The following function was developed based on the described operations, where the counters for each iteration (in terms of robot movement) were acquired empirically.

```python
def motor_control_based_on_lidar(lidar_data, motor_left, motor_right,
    speed):
    global ready_rotate, ready_forward, m_counter, m_per_counter
    global rotation_angle, largest_gap_distance, largest_gap_index, rotation_counter

    m_per_counter = speed/2500 # estimated value
    rad_per_counter = (speed/(1.5*9.53))*np.pi/90 # estimated value
    precision = 0.84

    if not ready_rotate and not ready_forward:
        largest_gap_distance, largest_gap_index, rotation_angle = find_largest_gap(lidar_data)

        ready_rotate = True
        ready_forward = True
        m_counter = 0
        rotation_counter = 0

    if ready_rotate:
        # Activate motors to initiate rotation
        if rotation_angle > 0:  # Turn left
            turn_left(motor_left, motor_right, speed)
        elif rotation_angle < 0:  # Turn right
            turn_right(motor_left, motor_right, speed)

        # Check if the wheels has traveled the required distance
        traveled_distance = rotation_counter*rad_per_counter
        rotation_counter += 1

        if rotation_angle < 0:
            if abs(traveled_distance) >= abs(rotation_angle + 2*np.pi/3)*precision:
                # Stop both motors
                stop(motor_left, motor_right)
                ready_rotate = False
        elif abs(traveled_distance) >= abs(rotation_angle - 2*np.pi/3)*precision:
            # Stop both motors
            stop(motor_left, motor_right)
            rotation_counter = 0
            ready_rotate = False

    elif ready_forward :
        # Activate motors to start moving forward
        forward(motor_left, motor_right, speed)

        # Check if the wheels has traveled the required distance
        traveled_distance = m_counter*m_per_counter

        if flags["debug"]:
            print("distances forw", traveled_distance, largest_gap_distance)

        if abs(traveled_distance) >= abs(largest_gap_distance)*precision + 0.07:
            # Stop both motors
            stop(motor_left, motor_right)
            ready_forward = False
        m_counter += 1

    else:
        ready_forward = False
        ready_rotate = False
        stop(motor_left, motor_right)
```

3 The bubble!

With the algorithm developed, the labyrinth can be completed perfectly, but we hit the wall numerous times due to imperfections in it. Some hypotheses for these effects are the presence of imperfections due to numerical error, others due to human error in the development of the algorithm, but mainly due to the absence of the concept of a **protective bubble**. At this point, in order to finalise the route, some factor adjustment variables have already been used and act as an artificial bubble, for example, the very use of only 84% of the actual distances measured when deciding to stop the car or finish a rotation.

According to the literature, the safety bubble algorithm can be explained in 4 steps, as follows:

1. Find the nearest LIDAR point and put a "safety bubble" around it of radius r
2. Set all points inside the bubble to distance 0. All nonzero points are considered "free space"
3. Find the maximum length sequence of consecutive non-zeros among the 'free space' points - The max-gap
4. Find the "best" point among this maximum length sequence

However, due to the deadline a simplified version was explored, in which the safety bubble technique in the follow gap algorithm, was made by adjusting of the movement behaviour to maintain a safe distance from obstacles detected by the LIDAR sensor. So for the nearest point, the following function was used:

```
def find_nearest_lidar_point(lidar_data):

    # Initialize variables to store information about the nearst point
    nearest_lidar_point_index = -1
    nearest_lidar_point_distance = 0

    # Iterate through the points to find the nearest one
    for i in range(len(lidar_data)):
        dist = distance(lidar_data[i], [0, 0])

        # Update information if the current point is closer than the
   last one
```

```python
        if dist < nearest_lidar_point_distance:
            nearest_lidar_point_distance = dist
            nearest_lidar_point_index = i

    return nearest_lidar_point_index
```

Then, a short radius was used just because of simplicity:

```python
    lidar_data[nearst_point_idx - 1] = [0,0]
    lidar_data[nearst_point_idx] = [0,0]
    lidar_data[nearst_point_idx + 1] = [0,0]
```

Followed by the previously presented function for finding the largest gap. From that function, it was possible to create another one to find the "best" point, which here is considered as the furthest, as shown in the following function:

```python
def find_best_point(lidar_data, largest_gap_index):
    dist_1 = distance(lidar_data[largest_gap_index], [0, 0])
    dist_2 = distance(lidar_data[largest_gap_index + 1], [0, 0])

    best_dist = 0

    if dist_1 > dist_2:
        best_idx = largest_gap_index
        best_dist = dist_1
    else:
        best_idx = largest_gap_index + 1
        best_dist = dist_2

    rotation_angle = angle(lidar_data[best_idx], [0, 0])

    return best_dist, best_idx, rotation_angle
```

The result was not much better than the first approach. However, even with this, we moved on to the next stage of deepening the code to try to remove the rotations and translations so that the robot could make curves. The developed approach takes into account the left and right regions of the lidar data, to find the centre of it. We use the centre as a target, so basically for each iteration, we look to go into the centre of the currently available path inside the labyrinth.

```python
        # Calculate the average position for left and right regions
        p_r = len(lidar_data[0]) // 4
        p_l = 3 * len(lidar_data[2]) // 4

        x_r = 0
        for idx, point in enumerate(lidar_data[0]):
            if idx == p_r:
                break
            x_r += point

        x_l = 0
        for idx, point in enumerate(lidar_data[2]):
            if idx == p_r:
                break
            x_l += point

        y_r = 0
        for idx, point in enumerate(lidar_data[0]):
            if idx == p_r:
                break
            y_r += point

        y_l = 0
        for idx, point in enumerate(lidar_data[0]):
            if idx == p_r:
                break
            y_l += point

        x_r_avg = x_r/ p_r
        y_r_avg = y_r / p_r

        x_l_avg = x_l/ (len(lidar_data[2]) - p_l)
        y_l_avg = y_l/ (len(lidar_data[3]) - p_l)

        # Calculate the central point between left and right regions
        x = (x_r_avg + x_l_avg) / 2
        y = (y_r_avg + y_l_avg) / 2
```

Based on the x and y computed before, we apply the following functions for calibrating the speed and the direction of rotation for each motor, respectively:

```python
def control_speed(x, y):
    # Compute the distance to reach the target
    dist = distance([x, y], [0, 0])

    # Empirical maximum distance (it isn't always true)
    max_dist = 0.12
    speed = 9.53*dist/max_dist

    # In case of saturation, use full-speed
    if speed > 9.53:
        speed = 9.53

    # Debug
    if flags["debug"]:
        print("Distance to destination: ", dist)
        print("Speed: ", speed)

    return speed

def control_motors(motor_left, motor_right, speed, angle):
    # Negative angle, turn left
    if angle < -0.1:
        turn_left(motor_left, motor_right, speed)

    # Positive angle, turn right
    elif angle > 0.1:
        turn_right(motor_left, motor_right, speed)

    # If the angle to target is very close to zero, go forward
    else:
        forward(motor_left, motor_right, speed)
```

4 Conclusions

In conclusion, the development of a follow-the-gap algorithm for navigating a robot through environments with obstacles marks a significant achievement. However, during the implementation and testing phases, several challenges were encountered, including instances where the robot erroneously faced walls.

One of the primary reasons for the robot facing walls intermittently could be attributed to the limitations of the follow-the-gap algorithm itself. The algorithm relies heavily on accurately detecting and following open spaces (or ”gaps”) between obstacles. However, in environments with complex geometries or narrow passages, the algorithm may struggle to distinguish between true gaps and areas where obstacles are nearby, leading to misinterpretations and potential collisions with walls. To address these issues and enhance the robustness of the navigation system, the introduction of safety bubbles into the algorithm was suggested. Safety bubbles serve as virtual buffers around the robot, defining a minimum distance that the robot should maintain from obstacles at all times. By incorporating safety bubbles into the follow-the-gap algorithm, the robot can proactively avoid collisions by detecting when it approaches too close to an obstacle and adjusting its trajectory accordingly. Unfortunately, this was not the case with the implementation designed in this work.

Implementing safety bubbles involves continuously monitoring the distance between the robot and nearby obstacles using sensor data, such as from LIDAR or ultrasonic sensors. When the distance falls below the predefined safety threshold, the robot can either slow down, stop, or adjust its direction to steer away from the obstacle and maintain a safe distance. Integrating safety bubbles into the follow-the-gap algorithm adds a layer of protection and ensures that the robot navigates through environments more cautiously and reliably, minimising the likelihood of collisions and enhancing overall performance and safety.

The best result was found with the last method, in which we evaluated the LIDAR data at each iteration and made a decision based on the angle value to reach the target in the middle of the best path. In future work, it would be interesting to improve the code with safety bubbles and also to explore the use of objects inside the maze to show the robustness of the controller. A video demonstration of the objective results is attached to this report.