Midterm Project Manual

GROUP 7

김송현 임승재 이소민

Midterm Project Manual

MODI Kit Instruction

- Used MODI Modules
- Model Design
- Used Algorithm for Moving

Code File Introduction

- Code Structure
- Explanation of each Function

Limitation & Possible Improvement

MODI Kit Instruction

Used MODI Modules

Model Design

Used Algorithms for Moving

1. Used MODI Modules

Moter & Moter Controller Module

Moter Controller Module : 2
Moter Module : 4
Wheels attached : 4

Used as Power Source

IR Module

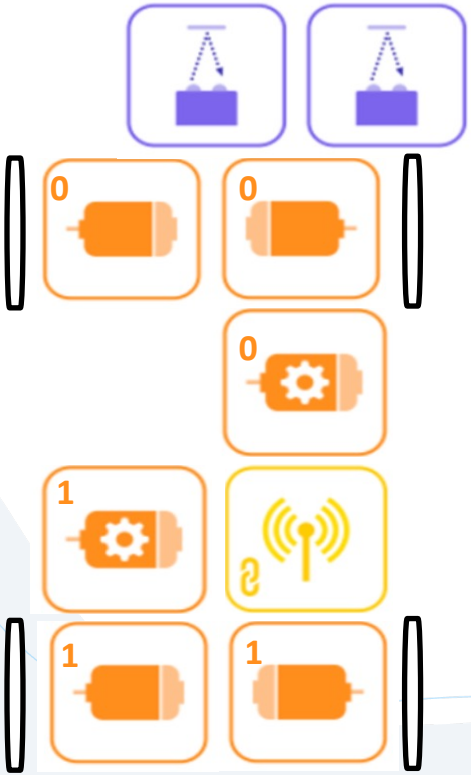
IR Module : 2

Used for Calibration

Network Module

Used for
Uploading code

2. Model Design



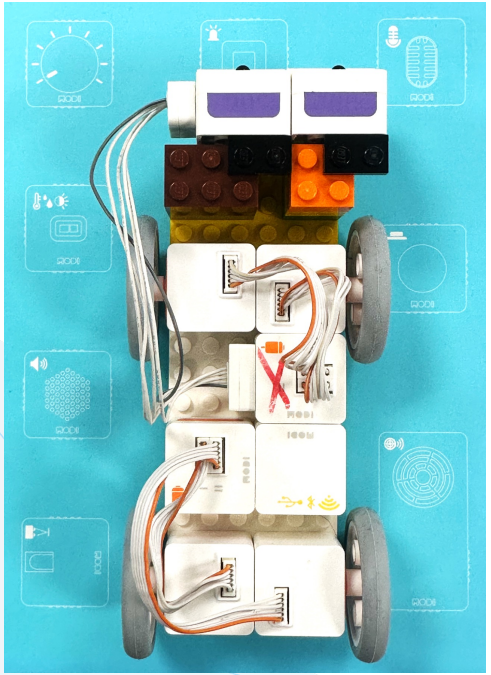
IR
Module

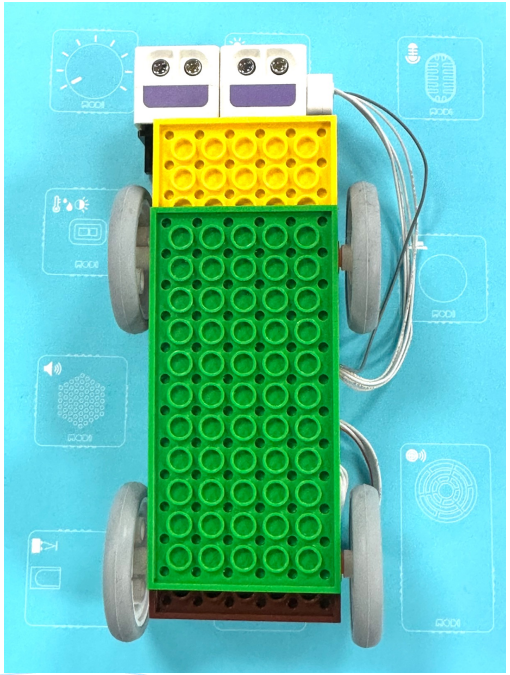
Moter
Module

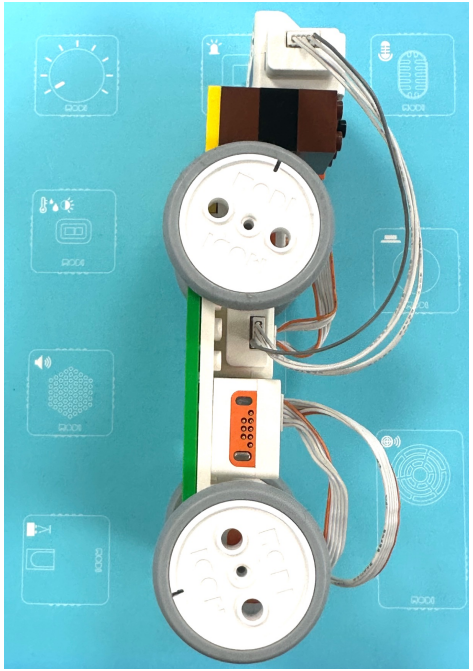
Moter Controller
Module

Network
Module

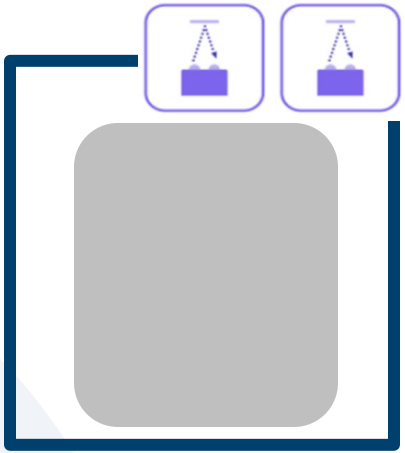
2. Model Design



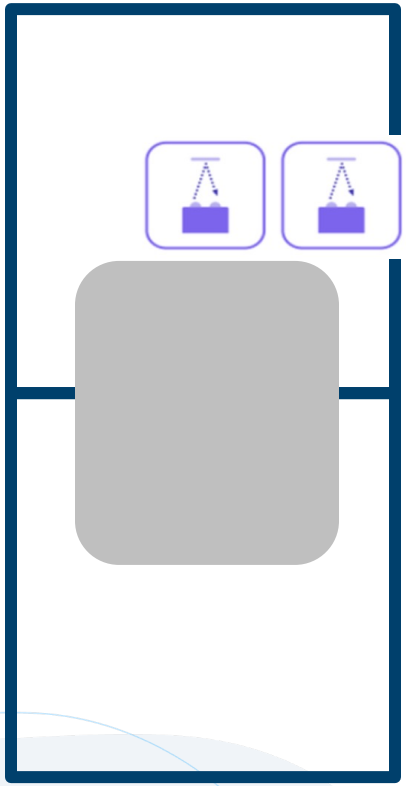


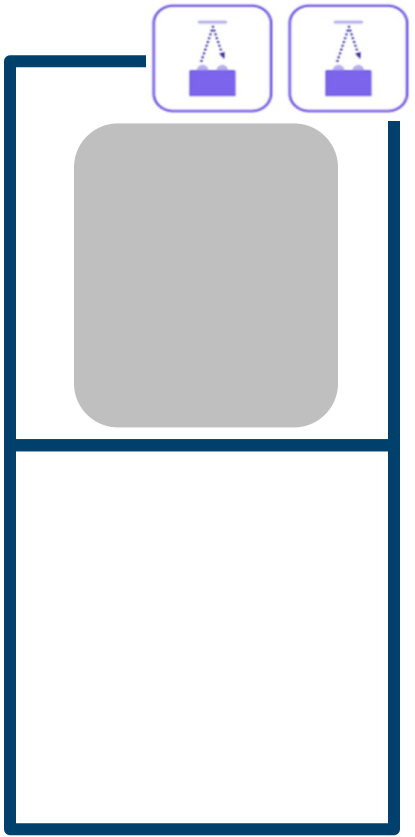


3. Used Algorithm for Moving

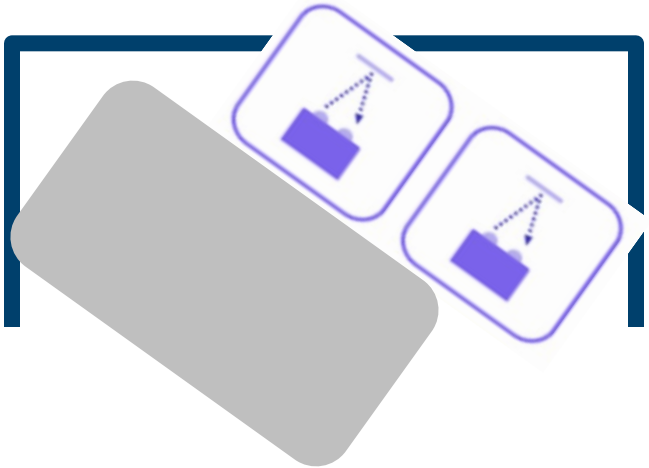


Go straight until
the model meets
the next black line





3. Used Algorithm for Moving



: Turn left for Calibration

Code File Introduction

Code Structure

Explanation of each function

Code Structure

```
import modi
import time
import numpy as np

def load_q_table(filename)


def move_forward(motors, irs)
def turn_right(motors, t)
def turn_left(motors,t)
def execute_moves(path, motors, irs)


def adam(cur, past, m_past, v_past, alpha, beta1, beta2, epsilon)


def make_move(current_pos, direction)
def move_to_goal(q_table, start_pos, goal_pos, alpha, beta1, beta2, epsilon)


def print_path(grid_size, path)
```

Load the Q-table from a text file

:param filename: str, the name of the file containing the Q-table
:return: list of lists, representing the Q-table

```
def load_q_table(filename):
    try:
        with open(filename, 'r') as file:
            lines = file.readlines()

        q_table = []

        for line in lines:
            # Only consider lines that contain a '[' character, indicating the start of a list
            if '[' in line:
                # Clean up the line by removing unwanted characters
                cleaned_line = line.replace('[', '').replace(']', '').replace('\n', '')

                # Split the cleaned line by spaces to get each value as a separate string
                row_strings = cleaned_line.split()

                # Convert each string value to a float and add the resulting list as a row in the Q-table
                q_table.append([float(value) for value in row_strings])

        return q_table

    except Exception as e:
        print(f"An error occurred while loading the Q-table: {e}")
        return None
```

Moves forward until an obstacle is detected with the IR sensors

:param motors: list of motor modules to control the robot
:param irs: list of IR sensor modules to detect obstacles

```
def move_forward(motors, irs):

    motors[0].speed = -50, 50
    motors[1].speed = -50, 50
    time.sleep(1.0)

    # check reach
    while(irs[0].proximity > 49):
        motors[0].speed = -50, 50
        motors[1].speed = -50, 50

    motors[0].speed = 0, 0
    motors[1].speed = 0, 0
    time.sleep(0.2)

    # car go forward but little move right, so if dont reach both end, correct it
    if irs[1].proximity > 49:
        turn_left(motors, 0.01)
```

Turns the robot left/right for a specified duration

:param motors: list of motor modules to control the robot
:param t: float, duration for which the robot turns left /right

```
def turn_right(motors, t):
    motors[0].speed = -50, -50
    motors[1].speed = -50, -50
    time.sleep(t)

def turn_left(motors,t):
    motors[0].speed = 50, 50
    motors[1].speed = 50, 50
    time.sleep(t)
```

Executes a sequence of moves to follow a given path

- param path: list of tuples, representing the coordinates of the path to follow
- param motors: list of motor modules to control the robot
- param irs: list of IR sensor modules to detect obstacles

```python
def execute_moves(path, motors, irs):
    facing = 'down'  # Initialize the starting direction
    move_commands = {
        (0, -1): 'left',
        (0, 1): 'right',
        (-1, 0): 'up',
        (1, 0): 'down'
    }

    for i in range(1, len(path)):
        current_pos = path[i - 1]
        next_pos = path[i]

        # Determine the direction of the next move
        move_vector = (next_pos[0] - current_pos[0], next_pos[1] - current_pos[1])
        chosen_direction = move_commands[move_vector]

        # Turn the car to face the chosen direction
        while facing != chosen_direction:
            if (facing == 'right' and chosen_direction == 'down') or \
               (facing == 'up' and chosen_direction == 'right') or \
               (facing == 'left' and chosen_direction == 'up') or \
               (facing == 'down' and chosen_direction == 'left'):
                turn_right(motors, 1.4)  # Assuming this function turns the car right
            else:
                turn_left(motors, 1.6)  # Assuming this function turns the car left

            # Stop motors for a moment after turning
            motors[0].speed = 0
            motors[1].speed = 0
            time.sleep(0.5)

            # Update the facing direction
            facing = chosen_direction

        # Move the car forward in the chosen direction
        move_forward(motors, irs)  # Assuming this function moves the car forward

        # Stop motors for a moment after moving
        motors[0].speed = 0
        motors[1].speed = 0
        time.sleep(1)
```

Simulate Adam optimizer step / Calculate new position

```python
def adam(cur, past, m_past, v_past, alpha, beta1, beta2, epsilon):
    """
    Simulate an Adam optimizer step for updating values.

    :param cur: list, current state values
    :param past: list, past state values
    :param m_past: list, past first moment estimates
    :param v_past: list, past second moment estimates
    :param alpha: float, learning rate
    :param beta1: float, exponential decay rate for the first moment estimates
    :param beta2: float, exponential decay rate for the second moment estimates
    :param epsilon: float, a small constant to avoid division by zero
    :return: list, updated state values
    """
    m_cur = [(beta1 * m_past_i + (1 - beta1) * (cur_i - past_i)) for m_past_i, cur_i, past_i in zip(m_past, cur, past)]
    v_cur = [(beta2 * v_past_i + (1 - beta2) * ((cur_i - past_i) ** 2)) for v_past_i, cur_i, past_i in zip(v_past, cur, past)]

    m_hat = [m_cur_i / (1 - beta1) for m_cur_i in m_cur]
    v_hat = [v_cur_i / (1 - beta2) for v_cur_i in v_cur]

    new_cur = [cur_i + alpha * m_hat_i / (np.sqrt(v_hat_i) + epsilon) for cur_i, m_hat_i, v_hat_i in zip(cur, m_hat, v_hat)]

    return new_cur

def make_move(current_pos, direction):
    """
    Calculates the new position after making a move in the given direction.

    :param current_pos: tuple, the current coordinates (x, y)
    :param direction: str, the direction to move in
    :return: tuple, the new position after the move
    """
    moves = {
        'left': (0, -1),
        'right': (0, 1),
        'up': (-1, 0),
        'down': (1, 0)
    }
    move = moves.get(direction, (0, 0))
    return (current_pos[0] + move[0], current_pos[1] + move[1])
```

Simulate Adam optimizer step – Explanation

```python
def adam(cur, past, m_past, v_past, alpha, beta1, beta2, epsilon):
    """
    Simulate an Adam optimizer step for updating values.

    :param cur: list, current state values
    :param past: list, past state values
    :param m_past: list, past first moment estimates
    :param v_past: list, past second moment estimates
    :param alpha: float, learning rate
    :param beta1: float, exponential decay rate for the first moment estimates
    :param beta2: float, exponential decay rate for the second moment estimates
    :param epsilon: float, a small constant to avoid division by zero
    :return: list, updated state values
    """
    m_cur = [(beta1 * m_past_i + (1 - beta1) * (cur_i - past_i)) for m_past_i, cur_i, past_i in zip(m_past, cur, past)]
    v_cur = [(beta2 * v_past_i + (1 - beta2) * ((cur_i - past_i) ** 2)) for v_past_i, cur_i, past_i in zip(v_past, cur, past)]

    m_hat = [m_cur_i / (1 - beta1) for m_cur_i in m_cur]
    v_hat = [v_cur_i / (1 - beta2) for v_cur_i in v_cur]

    new_cur = [cur_i + alpha * m_hat_i / (np.sqrt(v_hat_i) + epsilon) for cur_i, m_hat_i, v_hat_i in zip(cur, m_hat, v_hat)]

    return new_cur
```

$$m_0 = 0, v_0 = 0$$

$$m_{t+1} \leftarrow \beta_1 m_t + (1 - \beta_1)\nabla_\theta \mathcal{L}(\theta)$$ Momentum

$$v_{t+1} \leftarrow \beta_2 v_t + (1 - \beta_2)\nabla_\theta \mathcal{L}(\theta)^2$$ RMS Prop

$$\theta_j \leftarrow \theta_{j+1} + \frac{\epsilon}{\sqrt{v_{t+1}} + 1e^{-5}} m_{t+1}$$ RMS Prop + Momentum

Sequence of moves to reach the goal using the Q-table

```python
def move_to_goal(q_table, start_pos, goal_pos, alpha, beta1, beta2, epsilon):
    """
    Determines the sequence of moves to reach the goal using the Q-table.

    :param q_table: numpy.ndarray, the Q-table used for decision making
    :param start_pos: tuple, the starting position on the grid
    :param goal_pos: tuple, the desired goal position on the grid
    :param alpha: float, the learning rate
    :param beta1: float, the beta1 parameter for the Adam optimizer
    :param beta2: float, the beta2 parameter for the Adam optimizer
    :param epsilon: float, the epsilon parameter for the Adam optimizer
    :return: list of tuples, representing the path to the goal
    """
    current_pos = start_pos
    path = [current_pos]
    m_past = [0, 0, 0, 0]
    v_past = [0, 0, 0, 0]
    past_state_values = [0, 0, 0, 0]
    current_state_values = q_table[current_pos]

    while current_pos != goal_pos:
        new_state_values = adam(current_state_values, past_state_values, m_past, v_past, alpha, beta1, beta2, epsilon)

        # Update past moment vectors for the next iteration
        m_past = [alpha * m_past_i + (1 - alpha) * (new_i - cur_i) for m_past_i, new_i, cur_i in zip(m_past, new_state_values, current_state_values)]
        v_past = [beta2 * v_past_i + (1 - beta2) * ((new_i - cur_i) ** 2) for v_past_i, new_i, cur_i in zip(v_past, new_state_values, current_state_values)]

        max_action = np.argmax(new_state_values)
        action_map = {0: 'left', 1: 'down', 2: 'right', 3: 'up'}
        direction = action_map[max_action]

        # Move to the next position based on the direction chosen
        print(current_pos)
        past_state_values = q_table[current_pos]
        current_pos = make_move(current_pos, direction)
        current_state_values = q_table[current_pos]
        path.append(current_pos)

        print(action_map)
        print(current_state_values)
        print(new_state_values)
        if current_pos == goal_pos:
            break

    return path
```

Printing path on an 8x8 grid

Main function

```python
def print_path(grid_size, path):
    """
    Prints the path on an 8x8 grid.

    :param grid_size: int, the size of the grid
    :param path: list of tuples, the path to print
    """
    # Create an empty grid
    grid = [['0' for _ in range(grid_size)] for _ in range(grid_size)]

    # Fill in the path with asterisks
    for x, y in path:
        grid[x][y] = '*'  # Assumes that path is 0-indexed

    # Print the grid
    for row in grid:
        print(' '.join(row))

if __name__ == "__main__":
    # Example usage:
    q_table = load_q_table('Q_table.txt')
    bundle = modi.MODI()
    irs = [bundle.irs[0], bundle.irs[1]]
    motors = [bundle.motors[0], bundle.motors[1]]
    if q_table:
        q_table = np.array(q_table).reshape((8, 8, 4))  # Reshape to 8x8 grid with 4 actions
        start_pos = (0, 0)  # Starting at position (1, 1) in 1-indexed notation, (0, 0) in 0-indexed.
        goal_pos = (7, 7)  # Goal position (8, 8) in 1-indexed notation, (7, 7) in 0-indexed.

        # Parameters for the Adam optimizer
        alpha = 0.001
        beta1 = 0.9
        beta2 = 0.999
        epsilon = 1e-10

        path = move_to_goal(q_table, start_pos, goal_pos, alpha, beta1, beta2, epsilon)
        print_path(8, path)
        execute_moves(path, motors, irs)

        print(f"Path to goal: {path}")
    else:
        print("Failed to load the Q-table.")
```

Limitation & Possible Improvement

Of the Car's Calibration & Used Algorithms

1. Calibration Limitation & Possible Improvement

Limitation

The straight line shakes to the right, so only the turn left is corrected

Possible Improvement

Calibration to the right side can also be corrected by attaching a sensor (using 3 sensors)

Angle of rotation can also be applied to the protrusion using an IR sensor

2. Used Algorithm’s Limitation & Possible Improvement

Limitation

If two excessively different paths are determined, the momentum of the adam may cause an impossible path

A path with less rotation may be the best path to travel to 1x18x8 of a square, but if it departs/arrives from elsewhere, it does not guarantee that a path with less rotation is best

Possible Improvement

Check the arrival status with BFS

If the pass does not exist, Alpha and Beta Hyperparameters can be tuned to develop