American University of Sharjah

College of Engineering

Department of Computer Science and Engineering

Spring 2024

CMP 49410 - Intelligent Autonomous Robotics

Homework 3

Submitted By:

Ahmad Alsaleh - @00093749

& Ahmed Alabd Aljabar - @00092885

& Omar Ibrahim - @00093225

& Yousef Irshaid - @00093447

Submitted To: Dr. Michel Pasquier

Submission Date: 30th March 2024

Question 1

a) Active sensors are those that emit a source of energy and measure the reaction of the emission. An example of that could be an ultrasonic range sensor that emits sound waves and measures the time of flight in a way of finding the range (i.e., distance to an object). On the other hand, passive sensors rely on measuring energy sources already present in the environment. An example of that can be cameras that work by capturing the light intensities of the environment. Interestingly, a camera can also be used to determine range. The distinction is important due to the restriction of environmental and energy requirements. The energy emitted from active sensors will interact with the environment, which may not be appropriate (e.g, the case when a laser sensor is pointed to a person's eye). The energy emitted might also alter the readings of other sensors in the vicinity (e.g., multiple ultrasonic sensors) leading to some inaccuracies. On the other hand, passive sensors consume less energy but suffer from noise and signal problems. Active sensing can be defined as dynamically changing the position of a sensor to improve data gathering. For example, a camera mounted on a moving head where the position changes to constantly get a better view of the environment.

b) A car odometer is related to a wheel encoder as it measures the distance traveled by tracking the rotation and movement of the wheels. A typical resolution of such a sensor is 2000 increments per revolution. Interpolation using quadrature encoders can be used for a higher resolution and accuracy. However, wheel slippage and environmental conditions can hinder the accuracy of such a sensor. For example, the wheel may rotate more often on a wet floor while still not covering the same distance traveled on a dry floor. In both cases, the wheel encoder will have the same result, but it is clearly erroneous. Forward kinematics involves calculating the configuration coordinates given the inputs of a robot (e.g., speed and orientation). In the context of odometry, forward kinematics estimates the robot's position and orientation (from some starting position) based on the rotation and movement of its wheels or legs. For example, an e-puck robot turns and moves right. Based on the speed and orientation values on which the e-puck moved, an estimation of its end position and orientation can be obtained using forward kinematics, which would help in localization.

Question 2

a) The purpose of logical sensors is to abstract the perception of the robot, making it easier for robot control.

Pros:

1. The robot would be able to use multiple types of physical sensor without distinction.
2. The utility of the world model as a virtual sensor becomes possible due to the abstraction provided.
3. The physical redundancy added to the robot ensures fault tolerance. For example, if we use multiple ultrasonic sensors and one of them fails, the robot can still function.

Cons:

1. It is not straightforward to come up with weights for each sensor for an accurate reading.
2. Combining sensors might introduce processing overhead.

b) Yes, logical sensors can have different sensor modalities. For example, a robot might have an infrared sensor, a laser sensor, an ultrasonic sensor, and a computer vision-based camera sensor all for the purpose of detecting obstacles. This would mean that the logical sensor would result in one aggregated percept data. This is particularly useful in cases where some sensors are preferred over others, and you can specify the logical sensor when not to use a particular type. The combination we could use is as mentioned above (an infrared sensor, a laser sensor, an ultrasonic sensor, and a computer vision-based camera sensor). The con of this design is the added complexity in the integration of the sensors. For example, the utility of a selector function to select between the alternatives result in added complexity since we need to identify in which case a specific sensor serves the robot better. Furthermore, since physical sensors can have different resolutions and/or sensitivities and processing times, synchronization issues may arise. For example, it takes more time to process an image than a distance reading from a range sensor.

c) Sensor fusion can be defined as the process of combining sensor information into a single percept.

Pros:

1. One sensor alone can be imprecise or suffer from noise. Adding more sensors will help mitigate that issue and improve accuracy of the percept.

2. This follows from the concept of bagging in machine learning, where combining results of multiple models into one has been proven to give better results than the result of one meter model. Similarly, combining sensors will achieve the same effect.

3. In some implementations, sensors may provide mutually exclusive types of information about a percept. In such cases using sensor fusion provides a complete view about the percept. In this case the sensors are complementary to each other.

Cons:

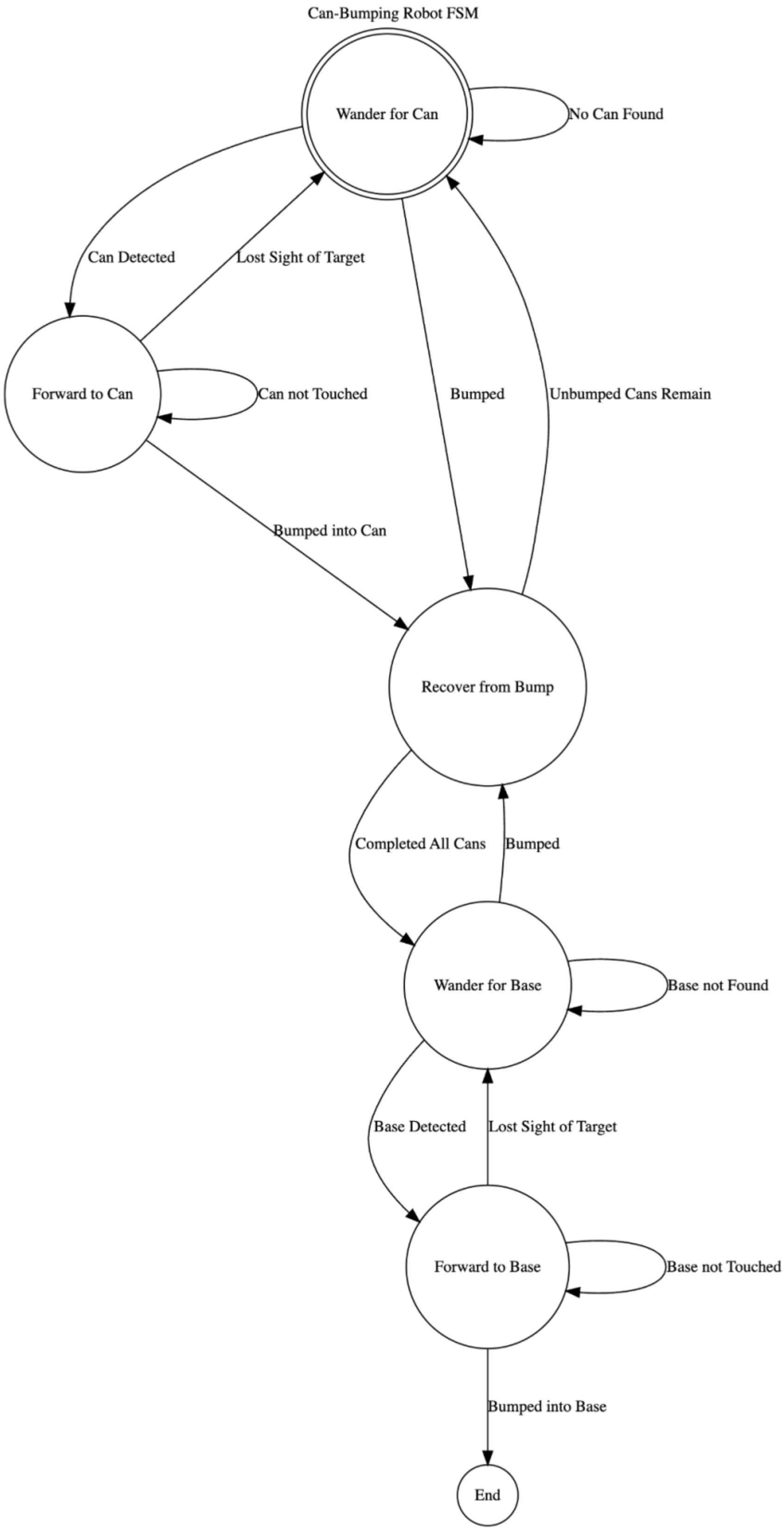
1. Sensors are redundant. In some cases, the cost is high, and this redundancy is not feasible.

2. Since physical sensors can have different resolutions and/or sensitivities and processing times, synchronization issues may arise.

In general, logical sensors use the concept of sensor fusion to abstract the hardware design. Sensor fusion can be physically redundant (i.e., same modality) or logically redundant (i.e., different modalities combined). In this case, hybrid logical sensors use logically redundant sensor fusion. Nevertheless, it is worth noting that logical sensors do not need to use sensor fusion and can instead rely on the world model the robot built as a source of information.

Question 3

a)


Can-Bumping Robot FSM
Wander for Can
No Can Found
Can Detected
Lost Sight of Target
Forward to Can
Can not Touched
Bumped
Unbumped Cans Remain
Bumped into Can
Recover from Bump
Completed All Cans
Bumped
Wander for Base
Base not Found
Base Detected
Lost Sight of Target
Forward to Base
Base not Touched
Bumped into Base
End

q	σ	D(q,σ)
Wander for can	Can detected	Forward to can
Wander for can	Bumped	Recover from bump
Wander for can	No Can Found	Wander for can
Forward to can	Bumped into can	Recover From bump
Forward to can	Lost sight of target	Wander for can
Forward to can	Can not Touched	Forward to can
Recover From bump	Unbumped cans remain	Wander for can
Recover From bump	Completed all cans	Wander for base
Wander for base	Bumped	Recover From bump
Wander for base	Base detected	Forward to Base
Wander for base	Base not Found	Wander for base
Forward to base	Lost sight of target	Wander for base
Forward to base	Base not Touched	Forward to base
Forward to base	Bumped into base	End

b) Code:

```
from controller import Robot, AnsiCodes
import numpy as np


class RobotState:
    WANDER_FOR_CAN = 1
    FORWARD_TO_CAN = 2
    WANDER_FOR_BASE = 3
    FORWARD_TO_BASE = 4
    RECOVER_FROM_BUMP = 5


class Colors:
    RED = 0
    GREEN = 1
    YELLOW = 2
```

```python
    BLUE = 3


class Controller(Robot):
    DETECTION_RATIO = 1.4
    MAX_WANDERING_COUNTER = 10
    MAX_SPEED = 7
    NUM_CYLINDERS = 3
    COLOR_NAMES = ["red", "green", "yellow", "blue"]
    ANSI_COLORS = [
        AnsiCodes.RED_FOREGROUND,
        AnsiCodes.GREEN_FOREGROUND,
        AnsiCodes.YELLOW_FOREGROUND,
        AnsiCodes.BLUE_FOREGROUND,
    ]
    RECOVERY_DURATION = 50

    def __init__(self):
        super().__init__()
        self.timeStep = int(self.getBasicTimeStep())

        self.state = None

        self.camera = self.getDevice("camera")
        self.camera.enable(self.timeStep)

        self.bumper = self.getDevice("bumper")
        self.bumper.enable(self.timeStep)

        self.left_motor = self.getDevice("left wheel motor")
        self.right_motor = self.getDevice("right wheel motor")
        self.left_motor.setPosition(float("inf"))
        self.right_motor.setPosition(float("inf"))
        self.left_motor.setVelocity(0.0)
        self.right_motor.setVelocity(0.0)

        self.left_speed = Controller.MAX_SPEED / 2
        self.right_speed = Controller.MAX_SPEED / 2
```

```python
        self.completed_cylinders = []
        # the COLOR of the currently-detected target
        self.current_target = None
        self.recovered_from = None

        self.wandering_counter = 0
        self.recovery_counter = 0

    def get_image_colors(self):
        """
            returns the summation of intensities in the 3 channels RGB
        """
        width = self.camera.getWidth()
        height = self.camera.getHeight()
        image = self.camera.getImage()
        red, green, blue = 0, 0, 0
        for i in range(int(width / 3), int(2 * width / 3)):
            for j in range(int(height / 2), int(3 * height / 4)):
                red += self.camera.imageGetRed(image, width, i, j)
                green += self.camera.imageGetGreen(image, width, i, j)
                blue += self.camera.imageGetBlue(image, width, i, j)
        return red, green, blue

    def bumped(self):
        return bool(self.bumper.getValue())

    def wander(self):
        """
            randomly chooses a value to change the motor speeds for a wandering behavior
            update every MAX_WANDERING_COUNTER cycles
        """
        if self.wandering_counter >= Controller.MAX_WANDERING_COUNTER:
            rand = np.random.uniform(-Controller.MAX_SPEED / 4, Controller.MAX_SPEED / 4)
            self.left_speed = np.clip(self.left_speed + rand, 0, Controller.MAX_SPEED / 2)
            self.right_speed = np.clip(self.right_speed - rand, 0, Controller.MAX_SPEED / 2)
            self.wandering_counter = 0
        else:
            self.wandering_counter += 1
```

```python
def detect_can(self):
    """
        detects can according to color and returns the color of the detected can.
        Return None if no can is detected
    """
    red, green, blue = self.get_image_colors()
    # If a color is much more represented than the other ones,
    # a cylinder is detected
    if (
        red > Controller.DETECTION_RATIO * green
        and red > Controller.DETECTION_RATIO * blue
    ):
        color = Colors.RED
    elif (
        green > Controller.DETECTION_RATIO * red
        and green > Controller.DETECTION_RATIO * blue
    ):
        color = Colors.GREEN
    elif (
        red > Controller.DETECTION_RATIO * blue
        and green > Controller.DETECTION_RATIO * blue
    ):
        color = Colors.YELLOW
    else:
        return None

    print(
        "Looks like I found a "
        + Controller.ANSI_COLORS[color]
        + Controller.COLOR_NAMES[color]
        + AnsiCodes.RESET
        + " cylinder"
    )
    return color

def detect_base(self):
    """
```

```
        Similar to detect_can, but detect a blue cube. Returns boolean (detected or not)
    """
    red, green, blue = self.get_image_colors()
    return (
        blue > (Controller.DETECTION_RATIO - 0.2) * red
        and blue > (Controller.DETECTION_RATIO - 0.2) * green
    )

def center_color(self, target_color):
    """
        The function return the motor speeds to center the robot on the target
        returns a tuple (can_see_target, left_speed, right_speed)
    """
    image = self.camera.getImage()
    width, height = self.camera.getWidth(), self.camera.getHeight()

    color_positions = []
    for x in range(width):
        for y in range(height):
            r = self.camera.imageGetRed(image, width, x, y)
            g = self.camera.imageGetGreen(image, width, x, y)
            b = self.camera.imageGetBlue(image, width, x, y)

            # Define simple thresholds for red, green, blue, and yellow
            # These thresholds are experimental
            is_target_color = False
            if target_color == Colors.RED and r > 175 and g < 50 and b < 50:
                is_target_color = True
            elif target_color == Colors.GREEN and g > 175 and r < 50 and b < 50:
                is_target_color = True
            elif target_color == Colors.BLUE and b > 100 and r < 50 and g < 50:
                is_target_color = True
            elif target_color == Colors.YELLOW and r > 175 and g > 175 and b < 50:
                is_target_color = True

            if is_target_color:
                color_positions.append(x)
```

```
        if not color_positions:
            return False, None, None

        # Find the average position of the detected color
        average_position = sum(color_positions) / len(color_positions)
        center_position = width / 2

        threshold = width * 0.05

        if average_position < center_position - threshold:
            # Color is to the left, rotate left
            left_speed = -1.0
            right_speed = 1.0
        elif average_position > center_position + threshold:
            # Color is to the right, rotate right
            left_speed = 1.0
            right_speed = -1.0
        else:
            # Color is centered, go forward
            left_speed = Controller.MAX_SPEED
            right_speed = Controller.MAX_SPEED

        return True, left_speed, right_speed

    def recover(self):
        """
            Recovery behavior if the robot bumps into something.
            Returns True if recovery is completed and False otherwise.
        """
        self.recovery_counter += 1
        if self.recovery_counter < Controller.RECOVERY_DURATION // 2:
            # reverse backwards for half a RECOVERY_DURATION
            self.left_speed = -Controller.MAX_SPEED / 2
            self.right_speed = -Controller.MAX_SPEED / 2
        elif (
            Controller.RECOVERY_DURATION // 2
            <= self.recovery_counter
            < Controller.RECOVERY_DURATION
```

```
    ):
        #  rotate for the other half of RECOVERY_DURATION
        self.left_speed = -Controller.MAX_SPEED / 4
        self.right_speed = Controller.MAX_SPEED / 4
    else:
        # reset the counter and signal the end of recovery
        self.recovery_counter = 0
        return True

    return False

def run(self):
    # starting state is wander for can
    self.state = RobotState.WANDER_FOR_CAN
    while self.step(self.timeStep) != -1:
        if self.state == RobotState.WANDER_FOR_CAN:
            """
            Wander until a target that has not been bumped into is detected.
            In that case transition to FORWARD_TO_CAN. If the robot bumps,
            transition to RECOVER_FROM_BUMP instead
            """
            self.wander()
            target = self.detect_can()
            if target is not None and target not in self.completed_cylinders:
                self.current_target = target
                self.state = RobotState.FORWARD_TO_CAN
            elif self.bumped():
                # the state is stored so that it can go back after recovery is done
                self.recovered_from = RobotState.WANDER_FOR_CAN
                self.state = RobotState.RECOVER_FROM_BUMP
        elif self.state == RobotState.FORWARD_TO_CAN:
            """
            In this state, center on the target and bump into it
            """
            see_target, left_speed, right_speed = self.center_color(self.current_target)
            if not see_target:
                self.state = RobotState.WANDER_FOR_CAN
                continue
```

```
            self.left_speed = left_speed
            self.right_speed = right_speed
            if self.bumped():
                # if bumped append the current target into an array and transition to RECOVER_FROM_BUMP
                self.state = RobotState.RECOVER_FROM_BUMP
                self.completed_cylinders.append(self.current_target)
                if len(self.completed_cylinders) == Controller.NUM_CYLINDERS:
                    # if we completed all cans, store WANDER_FOR_BASE so that we can go
                    # to it after recovery is completed
                    self.recovered_from = RobotState.WANDER_FOR_BASE
                else:
                    # if not completed all cans, then we go back to WANDER_FOR_CAN instead
                    # after recovery is completed
                    self.recovered_from = RobotState.WANDER_FOR_CAN

        elif self.state == RobotState.RECOVER_FROM_BUMP:
            recovered = self.recover()
            if recovered:
            # if recovery is done, go back to the state before recovery
                self.state = self.recovered_from
        elif self.state == RobotState.WANDER_FOR_BASE:
            """
            Wander until a base station (blue) is detected.
            In that case transition to FORWARD_TO_BASE. If the robot bumps,
            transition to RECOVER_FROM_BUMP instead.
            """
            self.wander()
            if self.detect_base():
                self.current_target = Colors.BLUE
                self.state = RobotState.FORWARD_TO_BASE
            elif self.bumped():
                # store WANDER_FOR_BASE so that we can come back after recovery
                self.recovered_from = RobotState.WANDER_FOR_BASE
                self.state = RobotState.RECOVER_FROM_BUMP
        elif self.state == RobotState.FORWARD_TO_BASE:
            # set the motor speeds to center on the base station
            see_target, left_speed, right_speed = self.center_color(self.current_target)
```

```
                if not see_target:
                    # if base station is no longer detected go back to wandering
                    self.state = RobotState.WANDER_FOR_BASE
                    continue
                self.left_speed = left_speed
                self.right_speed = right_speed
                if self.bumped():
                    # if robot bumps into base, then the mission is completed.
                    print('Mission Completed !!!!')
                    exit()



            self.left_speed = np.clip(
                self.left_speed, -Controller.MAX_SPEED, Controller.MAX_SPEED
            )
            self.right_speed = np.clip(
                self.right_speed, -Controller.MAX_SPEED, Controller.MAX_SPEED
            )

            self.left_motor.setVelocity(self.left_speed)
            self.right_motor.setVelocity(self.right_speed)


controller = Controller()
controller.run()
```