Intelligent Autonomous Robotics

Task 1

Robot avoids obstacles and follows walls

Julie Kew (s1676953)
Kurt Marcinkiewicz (s1670980)

Abstract

We have programmed a Khepera robot in Python to turn away from obstacles before contacting them and to follow walls at a consistent distance. Four IR sensors are used to detect obstacles and two to judge distance to walls. A subsumption control architecture gives precedence to obstacle avoidance when such are detected. The robot consistently follows walls in different light conditions and evades all but the most occluded of obstacles. Some limitations are that its obstacle avoidance mechanism prevents it from utilizing gaps under 11cm, its distance kept from walls depends upon the color and illumination of the wall and it has not been tested at greater than a snail's pace.

October 2016

1 Introduction

The instructions for this lab were to program a Khepera robot to drive around autonomously, avoid colliding with obstacles, avoid getting stuck and tend to follow walls. We accomplished this by using the front six IR sensors on the robot to detect walls and freestanding obstacles and storing within a robot object information about whether the robot was following walls or in the middle of a turn.

2 Methods

The robot uses a simple subsumption architecture with bottom-layer obstacle avoidance and second-layer wall following. It attempts to follow walls unless an obstruction is detected by its front four sensors, upon which it rotates in place to avoid the object and then continues wall following. A pseudocode outline follows.

```
Drive forward
loop
  Read IR
  if currently turning and not clear of obstruction then
    Continue turning
  else if obstruction detected then
    if following left wall then
      Turn right
    else if following right wall then
      Turn left
    else if obstruction on left then
      Turn right
    else if obstruction on right then
      Turn left
    end if
  else if abreast of wall then
    Follow wall
  else if following wall then
    Adjust wall distance
  else
    Drive forward
  end if
end loop
```

Information about what the robot is currently doing (following a left wall, following a right wall or turning) is stored in a robot object. When an obstacle is detected, the two drive motors are signaled the same speed in opposite directions to turn the robot in place, approximately 30° at a time.

If the left or right horizontal IR sensor (sensor 0 or 5) reads above 150, a wall is deemed close enough to follow. Thereafter, any reading between 200 and 300 is considered acceptable. If the reading is outside this range, the motors are run forward at slightly uneven speeds to cause the robot to veer closer to or farther from the wall.

In choosing this structure, we attempted to find the simplest method to meet the requirements. We first implemented obstacle avoidance with 90° left turns and no wall following. Then we added in the logic that if there was no obstacle to avoid, control should fall through to the wall following routine. We made various adjustments including shrinking the turning angle to make the robot more sensitive to the shapes of real objects and judging the turning direction based on prior knowledge of the world.

3 Results

When reaching an intersection while following a wall, the robot tended to turn and continue following the adjoining wall if the angle between the two walls was acute, right, or obtuse. If the angle was a reflex angle (particularly greater than about 200°), the robot would tend to continue into space instead of turning to follow the adjoining wall. Similarly, if approaching a round object, the robot would curve but tend not to follow the object all the way around.

When enclosed inside 4 walls with an exit that was less than or equal to about 11cm wide, the robot tended not to pass through the exit despite having enough room (the robot is about 7cm wide), instead circling within the enclosure indefinitely. If the exit was wider than about 11cm, the robot tended to pass through.

If the robot approached the end of a grey object such that less than 2cm of the object was obstructing the robot, the robot tended to bump into the object. This did not happen with bare wood objects.

4 Discussion

Using the Khepera's IR sensors, we were able to successfully navigate an environment, following along the surrounding walls, whilst avoiding obstacles of various shapes and materials. Using our approach, the robot was able to avoid getting stuck in dead ends of various sizes and angles. It was also able to maintain a consistent distance along walls.

Our approach did not facilitate the robot's passing through gaps less than or equal to 11cm wide, which the robot was small enough to fit through. This could be improved by better leveraging data we gathered about the range of readings from the IR sensors at various distances from different materials. Using this data, we could fine-tune the threshold of the side-facing sensors such that high readings from the side-facing sensors and low readings from the forward-facing sensors would indicate that the Khepera can continue forward.

The robot occasionally would bump into the ends of grey objects. Similar to above, we

can remedy this by fine-tuning the thresholds for the IR sensor readings.

A Source Code

Listing 1: main.py

```
#!usr/bin/env/python

from robot import Robot

import time


def main():
    robot = Robot()
    robot.go(4)

    try:
        while True:
            ir_result = robot.read_ir()

            if robot.continue_turning(ir_result) or robot.avoid_obstacle(
                    ir_result):
                time.sleep(.2)

            else:
                robot.set_following_wall(ir_result)

                if not robot.adjust_for_wall(ir_result):
                    robot.go(4)

                time.sleep(.025)

    except KeyboardInterrupt:
        robot.stop()


if __name__ == "__main__":
    main()
```

Listing 2: robot.py

```
from wall import Wall

import serial
```

```
class Robot:
    def __init__(self):
        self.conn = self._open_connection()
        self.following_wall = Wall.NONE
        self.ir_result = None
        self.__turning_to_evade = False

    CURVE_LEFT_VAL = (3, 4)
    CURVE_RIGHT_VAL = CURVE_LEFT_VAL[::-1]

    HARD_LEFT_VAL = (-2, 2)
    HARD_RIGHT_VAL = HARD_LEFT_VAL[::-1]

    OBSTACLE_READING_MIN = 120
    WALL_FOLLOWING_MIN = 150

    @property
    def turning_to_evade(self):
        return self.__turning_to_evade

    @turning_to_evade.setter
    def turning_to_evade(self, val):
        if val is True:
            self.following_wall = Wall.NONE

        self.__turning_to_evade = val

    def _open_connection(self, port="/dev/ttyS0", baudrate=9600,
                         stopbits=2,
                         timeout=1, **kwargs):
        s = serial.Serial(port=port, baudrate=baudrate, stopbits=stopbits,
                          timeout=timeout, **kwargs)

        if not s.isOpen():
            s.open()

        return s

    def _close_connection(self):
        if self.conn.isOpen():
            self.conn.close()

    def _send_command(self, command, verbose=True):
        # we should check if we have built up a backlog of serial messages
        # from the Khepera. If there is a backlog, we should read out of
        # the serial buffer so that future communications aren't messed up.
        if self.conn.inWaiting() > 0:
```

```
            if verbose:
                print "WARNING! Messages were waiting to be read!"
                print "This may be indicative of a problem elsewhere in your co

            while self.conn.inWaiting() > 0:
                message = self.conn.readline()[:-1]

                if verbose:
                    print message

        # we must make sure that the command string is followed by a newline
        # character before sending it to the Khepera
        if command[-1] != "\n":
            command += "\n"
        self.conn.write(command)

        # we check for a response from the Khepera
        answer = self.conn.readline()

        # now we can check if the response from the Khepera matches our
        # expectations
        if verbose:
            print "SENT     : " + command[:-1]
            print "RECEIVED : " + answer[:-1]

            if len(answer) < 1:
                print "WARNING! No response received!"

            elif answer[0] != command[0].lower():
                print "WARNING! Response does not match issued command!"

        return answer

    def _parse_sensor_string(self, sensor_string):
        if len(sensor_string) < 1:
            return -1

        else:
            # we need to remove some superfluous characters in the returned me
            print "sensor_string = " + sensor_string
            sensor_string = sensor_string[2:].rstrip(' \t\n\r')

            # and cast the comma separated sensor readings to integers
            sensor_vals = [int(ss) for ss in sensor_string.split(",")]

            return sensor_vals
```

```
def _set_speeds(self, left, right):
    # MAX = 127
    return self._send_command(
        "D," + str(int(left)) + "," + str(int(right)))

def read_ambient(self):
    ambient_string = self._send_command("O")

    return self._parse_sensor_string(ambient_string)

def read_ir(self):
    ir_string = self._send_command("N")

    return self._parse_sensor_string(ir_string)

def go(self, speed):
    return self._set_speeds(speed, speed)

def stop(self):
    return self.go(0)

def turn(self, left, right):
    return self._set_speeds(left, right)

def read_counts(self):
    count_string = self._send_command("H")

    return self._parse_sensor_string(count_string)

def set_counts(self, left_count, right_count):
    return self._send_command(
        "G," + str(left_count) + "," + str(right_count))

def set_wheel_positions(self, left_count, right_count):
    self.set_counts(0, 0)
    counts = self._parse_sensor_string(self._send_command("H"))

    return self._send_command(
        "C," + str(counts[0] + left_count) + "," + str(
            counts[1] + right_count))

def avoid_obstacle(self, ir_result):
    left_front_sensors = ir_result[1:3]
    right_front_sensors = ir_result[3:5]

    def stop_following_and_evade(turn):
```

```
            self.turning_to_evade = True
            self.turn(*turn)

        left_front_sensors_read_obstacle = any(
            [sensor > self.OBSTACLE_READING_MIN for sensor in
             left_front_sensors])
        right_front_sensors_read_obstacle = any(
            [sensor > self.OBSTACLE_READING_MIN for sensor in
             right_front_sensors])

        if left_front_sensors_read_obstacle or \
                right_front_sensors_read_obstacle:
            if self.following_wall == Wall.LEFT:
                stop_following_and_evade(self.HARD_RIGHT_VAL)

                return True

            elif self.following_wall == Wall.RIGHT:
                stop_following_and_evade(self.HARD_LEFT_VAL)

                return True

            if left_front_sensors_read_obstacle:
                stop_following_and_evade(self.HARD_RIGHT_VAL)

                return True

            if right_front_sensors_read_obstacle:
                stop_following_and_evade(self.HARD_LEFT_VAL)

                return True

        return False

    def adjust_for_wall(self, ir_result):
        left_sensor_reading = ir_result[0]
        right_sensor_reading = ir_result[5]
        min_threshold = 200
        max_threshold = 300

        if self.following_wall == Wall.LEFT:
            if left_sensor_reading > max_threshold:
                self.turn(*self.CURVE_RIGHT_VAL)

                return True

            if left_sensor_reading < min_threshold:
```

```
                self.turn(*self.CURVE_LEFT_VAL)

                return True

        if self.following_wall == Wall.RIGHT:
            if right_sensor_reading > max_threshold:
                self.turn(*self.CURVE_LEFT_VAL)

                return True

            if right_sensor_reading < min_threshold:
                self.turn(*self.CURVE_RIGHT_VAL)

                return True

        return False

    def set_following_wall(self, ir_result):
        if ir_result[0] > self.WALL_FOLLOWING_MIN:
            self.following_wall = Wall.LEFT

        elif ir_result[5] > self.WALL_FOLLOWING_MIN:
            self.following_wall = Wall.RIGHT

    def continue_turning(self, ir_result):
        if self.turning_to_evade and any(
                [sensor > self.OBSTACLE_READING_MIN for sensor in
                 ir_result[1:5]]):
            return True

        self.turning_to_evade = False
        return False
```

Listing 3: wall.py

```
from enum import Enum

Wall = Enum(NONE=0, LEFT=1, RIGHT=2)
```

Listing 4: enum.py

```
def Enum(**enums):
    return type('Enum', (), enums)
```