PROJECT ALTAIR

Week 2

Hasin Mahtab Alvee
210042174
SWE'21

Contents

1 Python Programming and OpenCV **1**
1.1 Python Programming 1
1.2 OpenCV 3

2 ROS (Robot Operating System) **7**
2.1 **Publisher and Subscriber nodes** 7
2.2 **Closed loop turtle bot simulation** 9

3 Theoretical Part **13**
3.1 Task 01 13
3.2 Task 02 14
3.3 Task 03 16

Link for Project Altair Week 2 Git Repo

Link for ROS Packages Git Repo

1 Python Programming and OpenCV

1.1 Python Programming

Logical 1. Progress Rover Instructions

A rover takes instructions and the grid size as input and then determines where on the grid it should be positioned.

```
def process_instructions(instructions, grid_size):
    rows = grid_size[0]
    columns = grid_size[1]

    directions = {
        'N': (0, 1),
        'E': (1, 0),
        'S': (0, -1),
        'W': (-1, 0)
    }
```

From the grid size tuple, we find the number of rows and columns of that grid. Then we initiate a dictionary of directions which keeps track of which way in the x and y axes the rover will move on a given instruction at a specific direction.

```
     x, y = 0, 0
    direction = 'N'

    turns = {'N': 0, 'E': 1, 'S': 2, 'W': 3}
```

We initialize the rover at (0,0) position, facing North. We initiate another dictionary with a key-value pair of the four directions.

```
    def Turn(direction, turnDirection):
        directionValue = turns[direction]

        turnOperations = {'L': -1, 'R': 1}

       newDirectionValue = (directionValue +
                            turnOperations[turnDirection]) % 4

        newDirection = next(
        key for key, value in turns.items() if value == newDirectionValue)

        return newDirection
```

Here, we define the Turn function, which will take input the current direction and the turn direction and then turn accordingly. We store the direction value from the turns dictionary using the current direction key. For Right, it adds 1 to the value of the current direction, and subtracts 1 for Left turn and mods the value with 4 so it can get reset every time the value is 4 or 0, so that we can go around in a circular way. Lastly, it takes the value and returns the direction from the dictionary.

```
    for i in instructions:
        if i == 'F':
            moveX = directions[direction][0]
            moveY = directions[direction][1]

            newX = x + moveX
            newY = y + moveY

            if 0 <= newX < rows and 0 <= newY < columns:
                x = newX
                y = newY

        elif i == 'L' or i == 'R':
            direction = Turn(direction, i)

    return x, y, direction
```

Now we loop over the instructions to check for R,L and F. If forward, we just add the x and y values from the directions dictionary for the current direction values, if the values are within the grid, only then we replace the old values with the new values otherwise we don't.

If, R and L is found, we call the Turn function and provide the current direction and the instruction turn R or L. Lastly, we return the x and y axes and the direction the rover is facing. We call the function as follows

```
if __name__ == "__main__":

    instructions = "FFLFFRFL"
    grid_size = (5, 5)
    print(process_instructions(instructions, grid_size))
```

References

1. Python Lists and Tuples
2. Dictionary in Python

1.2 OpenCV

Logical 2. OpenCV to Detect colors and extract data

Link for OpenCV Color detection

OpenCV is an open-source computer vision library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in commercial products. It is widely used to process images and videos using python.

Detect Colors

Here, we are given an image from which we need to detect the red and white colors and then analyze the image as an array to print the values of different pixel stats in the given image. First, we need to import OpenCV as cv2 and numpy as np. And we also need to import the Image class from the PIL library to access the image and convert it into a numpy array.

```
import numpy as np
import cv2

from PIL import Image
```

First, we work on the function to detect the red and white areas in the image, which takes an image as an input and outputs the different colored areas as images.

```
def detect_red_and_white_regions(image):
    img = cv2.imread(image, -1)
    smallImg = cv2.resize(img, (0, 0), fx=0.8, fy=0.8)
    cv2.imshow("Original-Small", smallImg)
```

Here, using the CV2 library, we take the image as an input and save it to an img variable. We resized the image just for the purpose of making it fit the viewport easily. And we show the original image with the imshow method.

```
    hsvImg = cv2.cvtColor(smallImg, cv2.COLOR_BGR2HSV)

    red_lower = np.array([136, 87, 111], dtype=np.uint8)
    red_upper = np.array([178, 255, 255], dtype=np.uint8)
    red_mask = cv2.inRange(hsvImg, red_lower, red_upper)
    res_red = cv2.bitwise_and(smallImg, smallImg, mask=red_mask)
    cv2.imshow("Red", res_red)
```

As images in CV2 are in BGR format, we need to change them to hsv format to be able to work on them for detecting colors, as hsv is far better capable of detecting objects and

colors than BGR or RGB.

For the Red colors in the image, we take a light red color value and a dark red color value in hsv format, and we apply a mask over the parts where the image is in range of the two red colors. Then we perform a bitwise AND operation on the image and the mask image. Only if there is a true pixel value in both the mask and the image, we return a true value from the operation. And this separates the red areas in the image and everything else has no pixel, so is dark. We do the same for the white areas and save it as res-white.

```
    white_red = cv2.addWeighted(res_red, 1.0, res_white, 1.0, 0.0)
    cv2.imshow("White_Red", white_red)

    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Then we use the addWeighted merge method and merge the two red and white images to make a complete red and white picture, and everything else is black, which means the other pixels have not been detected. And lastly, we close the window on any keypress to close the image viewers, and we move on to analyzing the image pixel array.

Analyze GOAT Method

The **analyze goat** method takes in the image as a numpy array and then computes different values.

```
def analyze_goat(image_array):
    max_value = np.max(image_array)
    min_value = np.min(image_array)
    print("Maximum pixel value: ", max_value)
    print("Minimum pixel value: ", min_value)

    average_pixel = int(np.mean(image_array))
    print("Average_pixel: ", average_pixel)

    shape = image_array.shape
    print("Shape of the image: ", shape)

    non_zero_pixels = np.count_nonzero(image_array)
    print("Total number of non-zero pixels: ", non_zero_pixels)

    zero_pixels = np.prod(shape) - non_zero_pixels
    print("Total number of zero pixels: ", zero_pixels)
```

First, we use the max and min functions from numpy to determine the maximum and minimum values of the array, which are respectively 255 and 0, which are the max and min pixel values of the image. The average pixel value can be found using the mean function in numpy. The shape of an image is the dimension of the array, which includes the height, width, and number of channels or colors in the image array. A built-in function can tell us how many non-zero pixels are there in the image, but to find the number of zero-pixels, we must subtract the number of non-zero pixels from the total number of pixels in an image array, which can be found from the product of the shape of the image.

```
if __name__ == '__main__':

    goat = "img/Goat.jpg"
    detect_red_and_white_regions(goat)

    img = Image.open("img/Goat.jpg")
    img_data = np.asarray(img)
    analyze_goat(img_data)
```

Lastly, we call the functions from the main method and then output the results in the openCV image viewer and terminal.

References

1. NumPy Tutorial
2. OpenCV Tutorial
3. Detecting Multiple Colors

2 ROS (Robot Operating System)

Practical 1. ROS Turtlesim

2.1 Publisher and Subscriber nodes

Publisher and subscriber nodes in ROS are the fundamental concepts of how all scripts work in ROS. The publisher acts as a piece of code that executes an operation, and the subscriber reads data from that operation. These scripts can be separate on their own or be in a closed loop connection, which is necessary if we want to read the data from the subscriber and adjust our publisher execution accordingly.

Link for Basic Publisher and Subscriber Node

Publisher

For this example, we will be using the built-in turtle simulator in ROS which will draw a circle.

```
#!/usr/bin/env python3

import rospy
from geometry_msgs.msg import Twist
```

First, we import the rospy and geometry libraries to send linear and angular messages, which will determine the path of the turtle to draw a cricle.

```
if __name__ == '__main__':
    rospy.init_node('draw_circle')
    rospy.loginfo("Started drawing circle")

    publisher = rospy.Publisher("/turtle1/cmd_vel", Twist, queue_size=10)

    rate = rospy.Rate(2)
```

In the main function, we initialize a new node by calling the init node method and define a name for the node. From the Publisher class, we create a publisher object, which takes three parameters. The first is which ROS Topic should use the scripts, then the datatype of that topic, and finally the queue size for how many actions to store. We also specify a rate in Hz at which the executor will sleep every second.

```
    while not rospy.is_shutdown():
        message = Twist()
        message.linear.x = 2.0
```

```
        message.angular.z = 1.0
        publisher.publish(message)
        rate.sleep()
```

Here, as long as the rospy doesn't receive any shutdown commands from the user, the while loop keeps working. Which creates a Twist type object as the input for the publisher and sets the x and z coordinates so it can draw a circle. Lastly, publish the message to execute it on the turtle.

Subscriber

For the subscriber, we will be reading the turtles current position on the x and y axis on the 2d grid.

```
#!/usr/bin/env python3

import rospy
from turtlesim.msg import Pose
```

We import Pose as this is the data type we need for the subscriber topic. The subscriber also uses a callback function to tell the topic what to do when the subscriber is executed.

```
def pose_callback(data: Pose):
    rospy.loginfo("x: %f, y: %f", data.x, data.y)


if __name__ == '__main__':
    rospy.init_node('turtle_pose_subscriber')
    subscriber = rospy.Subscriber(
        "/turtle1/pose", Pose, callback=pose_callback)

    rospy.loginfo("Started pose subscriber node")
    rospy.spin()
```

Here, the subscriber is first created with the Subscriber class, and it takes in the topic to utilize, the Pose data, and the callback function pose-callback. The function just takes in the Pose data and prints the x and y coordinates of the turtle at any given position. The spin method keeps the script alive until it is stopped.

2.2 Closed loop turtle bot simulation

Link for Loop system Publisher-Subscriber Node

The closed-loop publisher-subscriber design is very widely used as it lets us use the subscriber data that is returned by the publisher, and we can adjust the execution of the publisher according to the data.

A very common use case would be to avoid obstacles for autonomous robots in unknown environments. We take into account the data we get from the subscriber and then take that into consideration before executing the robot's next script. In this example, I've included three methods for the publisher to execute, taking into account the subscriber data.

```
#!/usr/bin/env python3

import rospy

from math import radians
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from turtlesim.srv import SetPen

previous_x = 0


def call_set_pen_service(r, g, b, width, off):
    rospy.wait_for_service("/turtle1/set_pen")
    try:
        set_pen = rospy.ServiceProxy("/turtle1/set_pen", SetPen)
        set_pen(r, g, b, width, off)
    except rospy.ServiceException as e:
        rospy.logerr(e)
```

Here, we import all the necessary libraries, and then we define a rosservice method, to better differentiate the turtle's track. The set-pen method sets the pen color to red when the turtle is going left and green when the turtle is going right. The variable previousX is also used for the set-pen method.

```
def pose_callback(pose: Pose):

    # avoid_border(pose)
    # draw_square()

    instructions = "FFFLFFFRFFFL"  # PLUS (+) Sign
    process_instructions(instructions)
```

```
if __name__ == '__main__':
    rospy.init_node('true_controller', anonymous=True)

    publisher = rospy.Publisher("/turtle1/cmd_vel", Twist, queue_size=10)

    subscriber = rospy.Subscriber(
        "/turtle1/pose", Pose, callback=pose_callback)

    rospy.loginfo("Started pose subscriber node")
    rospy.spin()
```

In the main method, we create the publisher and subscriber objects and then pass a callback function to the subscriber, which is essentially the publisher script. And we define the publisher function inside the pose-callback function.

```
def avoid_border(data):
    avoid_cmd = Twist()

    if data.x > 9.5 or data.x < 1.5 or data.y > 9.5 or data.y < 1.5:
        rospy.logwarn("Danger zone detected! Changing direction")
        avoid_cmd.linear.x = 1.0
        avoid_cmd.angular.z = - 1.3
    else:
        rospy.loginfo("Moving forward")
        avoid_cmd.linear.x = 2.0
        avoid_cmd.angular.z = 0.0

    publisher.publish(avoid_cmd)

    global previous_x
    if data.x >= 5.5 and previous_x < 5.5:
        rospy.logwarn("Crossed the middle line")
        call_set_pen_service(0, 255, 0, 5, 0)
    elif data.x <= 5.5 and previous_x > 5.5:
        rospy.logwarn("Crossed the middle line")
        call_set_pen_service(255, 0, 0, 5, 0)
    previous_x = data.x
```

The basic publisher method is to avoid the edges of the grid, which is done by comparing the x and y axes with the total grid, which is 11 points. We also set the pen to change color on the left and right sides. We create a twist object, which is the input to the publisher, and we change the x for the linear movement and the z for the angular turn. And it turns right every time it detects an edge.

```
def draw_square():

    rate = rospy.Rate(5)

    move_cmd = Twist()
    move_cmd.linear.x = 1.0

    turn_cmd = Twist()
    turn_cmd.linear.x = 0
    turn_cmd.angular.z = radians(45)

    while not rospy.is_shutdown():
        # Forward
        rospy.loginfo("Going Straight")
        for x in range(0, 10):
            publisher.publish(move_cmd)
            rate.sleep()

        # Turn 90 degrees
        rospy.loginfo("Turning")
        for x in range(0, 10):
            publisher.publish(turn_cmd)
            rate.sleep()
```

The draw-square is another basic method that draws a square when called. It uses the radians method to turn the turtle 90 degrees for every loop iteration, and the square is created.

```
def process_instructions(instructions):

    rate = rospy.Rate(5)

    move_cmd = Twist()
    move_cmd.linear.x = 2.0

    turn_R_cmd = Twist()
    turn_R_cmd.linear.x = 0
    turn_R_cmd.angular.z = radians(-45)

    turn_L_cmd = Twist()
    turn_L_cmd.linear.x = 0
    turn_L_cmd.angular.z = radians(45)

    for i in instructions:
        if i == 'F':
            # call_set_pen_service(255, 255, 255, 5, 0)
```

```
            rospy.logwarn("Moving forward")
            publisher.publish(move_cmd)
            rate.sleep()

        elif i == 'L':
            for x in range(0, 10):
                call_set_pen_service(255, 0, 0, 5, 0)
                rospy.logwarn("Turning Left")
                publisher.publish(turn_L_cmd)
                rate.sleep()

        elif i == 'R':
            for x in range(0, 10):
                call_set_pen_service(0, 255, 0, 5, 0)
                rospy.logwarn("Turning Right")
                publisher.publish(turn_R_cmd)
                rate.sleep()
```

Lastly, the final example is based on the first Python programming problem in the first Section. By inputting specific instructions to the turtle, it can move accordingly. Here, the instructions print a PLUS sign, and the lines are red when turning left and green when turning right.

References

1. Turtle BOT Guide

3 Theoretical Part

3.1 Task 01

Theoretical 1. LoRa Module for Communication

Communication to transfer data is very crucial between devices, such as IoT devices in remote rover deployments projects. Different types of wireless methods are used to receive data from devices, some of the most widely used are the Generic Radio Frequency module and the LoRa Module. Both have their advantages and disadvantages.

In a scenario, where a rover is deployed in a remote and vast area, the **LoRa connection module** would be more effective than a generic radio frequency module. Let's consider the use cases -

- **Data transmission Range :** For a vast area, the LoRa is better, as it is the best for long range connections. LoRa means, Long Range. It can be connected from 500 meters to all the way upto 10 km. Whereas, a generic RF module can only work correctly at maximum 25 meters.
- **Power Consumption :** Power consumption is lower in the generic RF module, which is only 2.9mA while receiving and only 0.3mA while sending. Compared to that, LoRa has a significant amount, which is 10mA both sending and receiving, which is still low than most other wireless communication methods.
- **Data transfer Rate :** LoRa sacrifices data transfer rate for the long range connectivity, though the generic RF module isn't any better than LoRa. The generic RF only has 4 kilobits per second transfer rate. Whereas, LoRa has a max data transfer rate of 37.5 kilobits per second which is great for transmitting sensor data, but video transmitting is not possible. It can transfer sensor data which can be used to measure environmental conditions such as temperature, sound, pollution levels, humidity and wind. Furthermore, the data transfer rate can be changed if wanted to increase the rate for a smaller cover range. Which makes it very adaptive to environments.
- **Connectivity in challenging environments. :** LoRa utilizes different types of gateways communication methods for faster and safer data transmission. Two architectures are - Gateway LoRa and Waspmote. Both of these make sure that data can be sent and received over larger areas. Which also makes it easier to maintain connectivity in challenging environments, which the generic RF module lacks.

References

1. LoRa Connectivity
2. NRF24 VS LoRa For Wireless Communication Between IoT Devices

3.2 Task 02

Theoretical 2. Pose Graph Optimization in SLAM

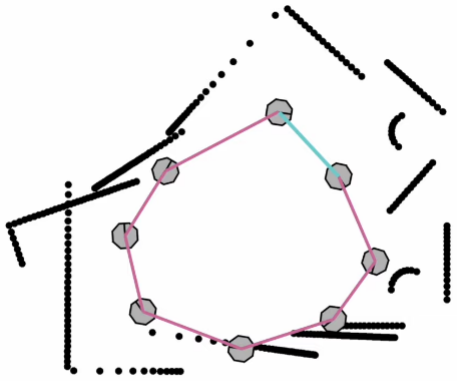
SLAM stands for Simultaneous Localization and Mapping, which means there will be no given data on the model of the environment. SLAM used two types of algorithm to map the environment,

- **Filtering :** Measures the latest movements on the go to update the environment map.
- **Smoothing :** Takes into account the complete set of robot trajectories to map the environment. And the recent factor standard framework in Pose graph optimization for this method.

In a scenario, a vehicle is equipped with LIDAR and odometry sensors to perceive its surroundings and estimate its position and orientation. The LIDAR helps to calculate the disrance between the vehicle and the obstacles, while the odometry uses movements of the encoder to determine how much the vehicle has moved from it's previous location.

For the ideal conditions, where the LIDAR and odometry will give precise locations and data, which in return will give us the perfect position for the map in one loop. And we can map the environment without any error. But this mapping example is not realistic. Both the LIDAR measurements and odometry will have it's errors and those error will add up for each loop if they are not considered and fixed with the iteration of loops in mapping.

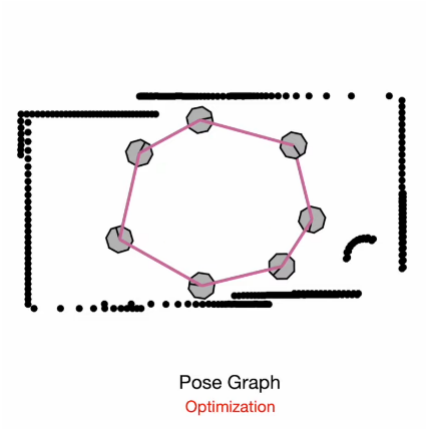
Pose graph optimization is a specific framework within SLAM that optimizes the estimated poses (positions and orientations) of the robot and the mapped environment based on sensor measurements. It involves creating a graph where robot poses are represented as nodes, and constraints between these poses are represented as edges. Using the pose graphing method, we can save the readings of each time the robots moves and measures the distance of the next obstacle or wall, we save that to a graph and continue to do this until the robot comes back to it's starting position. But we still can't do much with these readings and that's when the edges come in handy.



Pose Graph

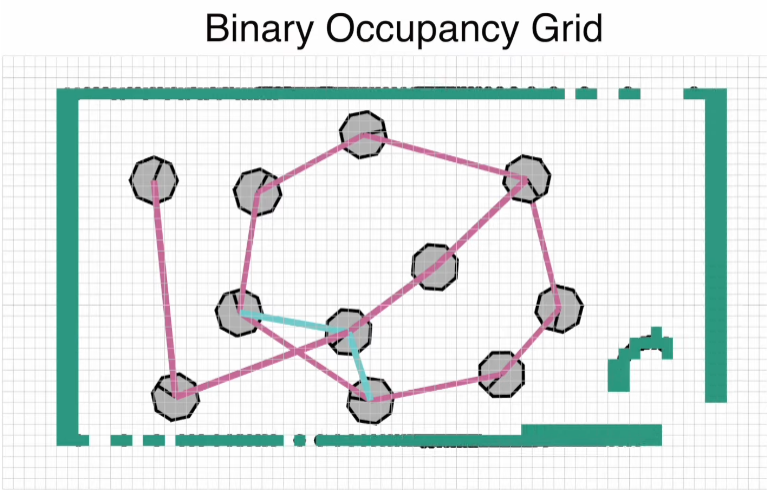
As we have two positions of the robot with the exact same obstacle data, it means those two positions are the same, so any difference between the two points that are joined by the

edge, can be dismissed and should be put onto each other, as this creates a tension among all the other connected edges and nodes as well, this pulls the graph closer and giving it a better shape which is similar to the actual map.



As we keep looping around the map and keep measuring data and the distances between nodes and keep track of the edges, we can get the almost exact map data only after a few more iterations of the whole environment. Even if some incorrect measurements are made, they can always be fixed using the other nodes as this Pose graph method uses absolute positioning instead of relative positioning.

Now that we have out Pose graph ready, we can turn this into an environment map model using different methods, such as Binary Occupancy Grid and Probabilistic Occupancy Grid. The binary method uses 0 and 1 to represent if a grid position is occupied or not and then builds a map around the occupied points. And the Probabilistic takes into account how much a point in the grid is occupied between 0 and 1, and then have black cells for occupied and white cells for un-occupied space, and everything else is in gray.



3.3 Task 03

Theoretical 3. Building an autonomous rover with Pixhawk

Pixhawk is an open-source autopilot hardware and software platform designed for drones. It serves as the flight controller and navigation system for these drones. Pixhawk is equipped with sensors such as accelerometers, gyroscopes, magnetometers, and GPS receivers to gather data about the drone's orientation, position, and motion. However, the exact same hardware firmware software stack can be used on different vehicle types like autonomous boats, rovers, submarines, etc.

To start with a smart autonomous rover, we first need to configure the hardware with the Pixhawk.

- A rover for autonomous is best suited with the Conventional steering and the Crawler model, which can get up to 8 to 10 MpH on any kinds of roads. Conventionally best size would be to make it 1/10 or 1/8.
- We will use a 4GB Micro SD with the Phixhawk, a wifi telemetry, which will directly be plugged into the Pixhawk LAN01.
- We will use a mini servo to hold the steering rod. Using a servo to connect with the wheels for steering, connect the servo to the AUX out, so the Pixhawk can control the servo.
- A ppm encoder will be connected to the RC pin of the Pixhawk. We will connect the GPS wire to the Pixhawk GPS port and the external compass to the I2C port.
- We connect the motor ESC to the power module for a static connection and connect the power module to the Pixhawk power port, so the Pixhawk can get power from the battery through the power module.
- Next we need to connect the Pixhawk to a Micro Controller in a UART connection. Once that is connected, we can connect a webcam to the connection.

Once we are done with the hardware, we need to configure the Pixhawk firmware for it to be compatible with Rover models.

- First we flash the Pixhawk with firmware from the Mission Planner. We need to specify the rover firmware with the Pixhawk1 controller and upload the firmware.
- Next we need to set up the python scripts to run on the Pixhawk rover, and we can set the start and ending point that would be determined via the GPS module.

References

1. Pixhawk Tutorial