SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 2nd semester

Introduction to Robotics Automation (PGI20D08J)- Lab Manual

This manual provides a structured guide for practical exercises in Robotics Automation, covering topics from Raspberry Pi fundamentals to advanced control and machine learning applications in robotics. Each lab outlines the objectives, step-by-step procedures, and sections for code, input, and expected output.

Lab 1: Introduction to Raspberry Pi and GPIO Programming

Title

Introduction to Raspberry Pi - Setup, OS, and GPIO Programming

Aim

To familiarize with the Raspberry Pi setup, its operating system, and to perform basic GPIO programming for interfacing sensors and actuators.

Procedure

1. **Setting up Raspberry Pi:**
   - Connect the necessary peripherals (monitor, keyboard, mouse).
   - Insert the SD card with the Raspberry Pi OS image.
   - Power on the Raspberry Pi.
   - Complete the initial setup wizard (locale, password, network).
2. **Exploring Raspberry Pi Operating System:**
   - Navigate the desktop environment.
   - Open the terminal and execute basic Linux commands (e.g., `ls`, `cd`, `pwd`, `sudo apt update`, `sudo apt upgrade`).
   - Understand the file system structure.
3. **GPIO Programming with Raspberry Pi:**
   - Identify the GPIO pins on the Raspberry Pi board.
   - Install necessary Python libraries for GPIO control (e.g., `RPi.GPIO`).
   - Connect a simple LED circuit to a GPIO pin with a current-limiting resistor.
   - Write a Python script to blink the LED.
   - Connect a push button to another GPIO pin and write a script to detect button presses and control the LED.
   - (Optional) Interface a simple sensor (e.g., a passive infrared (PIR) sensor) and read its output.

Source Code

```
# Example Python script for LED blinking
import RPi.GPIO as GPIO
import time

# Set up GPIO mode and warnings
GPIO.setmode(GPIO.BCM) # Use Broadcom pin-numbering scheme
GPIO.setwarnings(False)

# Define the GPIO pin for the LED
LED_PIN = 17

# Set the LED pin as an output
GPIO.setup(LED_PIN, GPIO.OUT)

print("LED blinking started. Press Ctrl+C to stop.")

try:
    while True:
        GPIO.output(LED_PIN, GPIO.HIGH) # Turn LED on
        print("LED ON")
        time.sleep(1) # Wait for 1 second
        GPIO.output(LED_PIN, GPIO.LOW) # Turn LED off
        print("LED OFF")
        time.sleep(1) # Wait for 1 second

except KeyboardInterrupt:
    print("\nLED blinking stopped.")
finally:
    GPIO.cleanup() # Clean up GPIO settings
```

Input

No direct user input for the LED blinking script. For the button example, the input would be a physical button press.

Expected Output

- Successful boot into Raspberry Pi OS.
- LED connected to GPIO pin 17 will blink on and off every second.
- Terminal output will show "LED ON" and "LED OFF" alternating every second.
- (For button example) LED will turn on/off based on button presses, and corresponding messages will appear in the terminal.

Lab 2: Python Programming for Robotics Applications

Title

Controlling Simple Robots and Sensor Data Acquisition with Python

Aim

To develop Python programs for controlling simple robots (e.g., line-following robots) and to acquire and process data from sensors.

Procedure

1. **Robot Control with Python and Raspberry Pi:**
   - Understand the basic mechanics of a simple robot (e.g., a two-wheeled robot with DC motors).
   - Identify the motor driver (e.g., L298N) and its connection to Raspberry Pi's GPIO pins.
   - Write a Python script to control motor direction and speed (e.g., forward, backward, turn left, turn right, stop).
   - Implement a basic line-following algorithm using infrared (IR) sensors.
   - Integrate sensor readings with motor control logic to make the robot follow a line.
2. **Data Acquisition and Processing from Sensors:**
   - Connect an analog sensor (e.g., a potentiometer or an analog temperature sensor) to an Analog-to-Digital Converter (ADC) module (e.g., MCP3008) interfaced with Raspberry Pi via SPI.
   - Write a Python script to read analog values from the sensor.
   - Process the raw sensor data (e.g., convert analog readings to meaningful units like temperature in Celsius).
   - Display the processed data in the terminal or log it to a file.

Source Code

```
# Example Python script for basic motor control (conceptual)
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

# Define motor control pins (example for a single motor)
MOTOR_IN1 = 23
MOTOR_IN2 = 24
ENABLE_PIN = 25 # For PWM speed control

GPIO.setup([MOTOR_IN1, MOTOR_IN2, ENABLE_PIN], GPIO.OUT)

# Set up PWM for speed control
pwm = GPIO.PWM(ENABLE_PIN, 100) # PWM frequency 100 Hz
pwm.start(0) # Start with 0% duty cycle (motor off)

def move_forward():
    GPIO.output(MOTOR_IN1, GPIO.HIGH)
    GPIO.output(MOTOR_IN2, GPIO.LOW)
    print("Moving Forward")

def stop_motor():
    GPIO.output(MOTOR_IN1, GPIO.LOW)
    GPIO.output(MOTOR_IN2, GPIO.LOW)
    print("Motor Stopped")

def set_speed(speed_percent):
    pwm.ChangeDutyCycle(speed_percent)
    print(f"Speed set to {speed_percent}%")

try:
    move_forward()
    set_speed(50) # Set speed to 50%
    time.sleep(3)
    stop_motor()
    set_speed(0) # Stop motor by setting speed to 0
    time.sleep(1)

except KeyboardInterrupt:
    print("\nProgram stopped.")
finally:
    pwm.stop()
    GPIO.cleanup()

# Example Python script for reading analog sensor data (conceptual for
MCP3008)
# Requires spidev library
# import spidev
# import time

# spi = spidev.SpiDev()
# spi.open(0,0) # Open SPI bus 0, device 0
# spi.max_speed_hz = 1000000 # 1MHz

# def read_adc(channel):
#     # ADC channel must be 0-7
#     if channel > 7 or channel < 0:
#         return -1
#     r = spi.xfer2([1, (8+channel)<<4, 0])
#     data = ((r[1]&3) << 8) + r[2]
#     return data

# try:
#     while True:
#         sensor_value = read_adc(0) # Read from channel 0
#         print(f"Raw sensor value: {sensor_value}")
#         # Convert raw value to meaningful units (e.g., voltage,
temperature)
#         # voltage = (sensor_value / 1023.0) * 3.3 # Assuming 3.3V reference
#         # print(f"Voltage: {voltage:.2f}V")
#         time.sleep(0.5)
# except KeyboardInterrupt:
#     spi.close()
#     print("\nSensor reading stopped.")
```

Input

- **Robot Control:** No direct user input for the basic script. For line following, the input is the line on the ground and readings from IR sensors.
- **Sensor Data Acquisition:** Physical changes in the environment affecting the sensor (e.g., light intensity for a photoresistor, temperature for a thermistor).

Expected Output

- **Robot Control:** The robot will move forward, stop, or follow a line as programmed. Terminal output will show corresponding movement commands.
- **Sensor Data Acquisition:** Raw sensor values and/or processed data (e.g., voltage, temperature readings) will be displayed in the terminal at regular intervals.

Lab 3: Implementing Basic Control Algorithms

Title

Implementing PID Control and Robot Communication with Raspberry Pi

Aim

To implement basic control algorithms like PID (Proportional-Integral-Derivative) control in Python and develop programs for robot communication and sensor data acquisition using Raspberry Pi.

Procedure

1. **Implementing PID Control in Python:**
   - Understand the theory behind PID control (P, I, D terms, error, setpoint, output).
   - Choose a simple system to control (e.g., a DC motor's speed, a robot's heading, or a temperature system).
   - Develop a Python class or function to implement the PID algorithm.
   - Integrate the PID controller with sensor feedback (e.g., encoder for motor speed, IMU for heading) and actuator control.
   - Tune the PID parameters (Kp, Ki, Kd) to achieve desired system response (e.g., minimal overshoot, fast settling time).
2. **Python Programs for Robot Communication, Sensor Data Acquisition, and Control:**
   - **Communication:** Establish serial communication (UART) or I2C/SPI between Raspberry Pi and another microcontroller or device. Write Python scripts to send commands and receive data.
   - **Sensor Data Acquisition:** Revisit sensor data acquisition from Lab 2, focusing on more complex sensors (e.g., IMU for orientation, ultrasonic sensor for distance). Implement robust data reading and filtering techniques.
   - **Integrated Control:** Combine sensor data acquisition, communication, and PID control to create a closed-loop system for a specific robot task (e.g., maintaining a constant speed, following a precise trajectory).

Source Code

```
# Example Python class for a simple PID controller
class PIDController:
    def __init__(self, Kp, Ki, Kd, setpoint):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.setpoint = setpoint
        self.prev_error = 0
        self.integral = 0
        self.output_limit = (-100, 100) # Example output limits

    def update(self, current_value, dt):
        error = self.setpoint - current_value

        self.integral += error * dt
        # Anti-windup for integral term
        if self.integral > self.output_limit[1] / self.Ki:
            self.integral = self.output_limit[1] / self.Ki
        elif self.integral < self.output_limit[0] / self.Ki:
            self.integral = self.output_limit[0] / self.Ki
        derivative = (error - self.prev_error) / dt
        self.prev_error = error

        output = (self.Kp * error) + (self.Ki * self.integral) + (self.Kd *
derivative)

        # Clamp output to limits
        output = max(self.output_limit[0], min(self.output_limit[1], output))
        return output

# Example usage (conceptual for motor speed control)
# pid = PIDController(Kp=1.0, Ki=0.1, Kd=0.05, setpoint=100) # Target speed
100 RPM
# current_speed = 0 # Assume this comes from an encoder
# dt = 0.1 # Time step

# while True:
#     control_signal = pid.update(current_speed, dt)
#     # Apply control_signal to motor PWM
#     # current_speed = read_motor_speed()
#     # time.sleep(dt)
```

Input

- **PID Control:** Real-time feedback from sensors (e.g., motor encoder readings, IMU data).
- **Robot Communication:** Commands sent from Raspberry Pi, data received from connected devices.

Expected Output

- **PID Control:** The controlled system (e.g., motor speed, robot heading) will converge to the setpoint with stable and desired response characteristics.
- **Robot Communication:** Successful exchange of data between Raspberry Pi and other devices.
- **Integrated Control:** The robot performs the specified task (e.g., maintains speed, follows a path) accurately and robustly due to closed-loop control.

Lab 4: Kinematics and Dynamics Simulation

Title

Robot Kinematics and Dynamics Simulation using V-REP/Gazebo

Aim

To use simulation software like V-REP (CoppeliaSim) or Gazebo to model and simulate robot kinematics and dynamics, and to implement Python scripts to interact with the simulation software for analysis.

Procedure

1. **Introduction to Simulation Software:**
   - Install and set up V-REP (CoppeliaSim) or Gazebo (with ROS).
   - Familiarize with the user interface, scene creation, and robot model importing.
2. **Modeling Robot Kinematics:**
   - Create a simple robotic arm model (e.g., a 2-DOF or 3-DOF manipulator) in the simulation environment.
   - Define joints, links, and coordinate frames.
   - Implement forward kinematics (FK) to calculate the end-effector position and orientation given joint angles.
   - Implement inverse kinematics (IK) to calculate joint angles required to reach a desired end-effector pose.
   - Test FK and IK by manipulating joint angles and observing end-effector movement, or by specifying end-effector poses and observing joint adjustments.
3. **Simulating Robot Dynamics:**
   - Assign physical properties (mass, inertia) to robot links.
   - Apply forces or torques to joints and observe the resulting motion.
   - Simulate gravity and collisions with the environment.
   - Analyze dynamic behavior, such as oscillations or stability.
4. **Python Scripting for Simulation Interaction:**
   - Learn how to use the remote API (for V-REP) or ROS topics/services (for Gazebo) to communicate with the simulation from Python.
   - Write Python scripts to:
     - Load/spawn robot models.
     - Read joint positions, velocities, and torques.
     - Set joint target positions or velocities.
     - Apply forces/torques.
     - Retrieve sensor data (e.g., vision sensor, force sensor).
     - Record and plot robot behavior data (e.g., joint trajectories, end-effector paths).

Source Code

```
# Example Python script for V-REP (CoppeliaSim) remote API interaction
(conceptual)
# Requires CoppeliaSim remote API client library
# import sim
# import time

# print('Program started')
# sim.simxFinish(-1) # close all opened connections
# clientID = sim.simxStart('127.0.0.1', 19999, True, True, 5000, 5) # Connect
to CoppeliaSim

# if clientID != -1:
#     print('Connected to remote API server')

#     # Get object handles
#     # errorCode, joint1_handle = sim.simxGetObjectHandle(clientID,
'Joint1', sim.simx_opmode_blocking)
#     # errorCode, joint2_handle = sim.simxGetObjectHandle(clientID,
'Joint2', sim.simx_opmode_blocking)

#     # Set joint target positions
#     # sim.simxSetJointTargetPosition(clientID, joint1_handle, 90 * math.pi
/ 180, sim.simx_opmode_oneshot)
#     # sim.simxSetJointTargetPosition(clientID, joint2_handle, 45 * math.pi
/ 180, sim.simx_opmode_oneshot)

#     # Start simulation (if not already running)
#     # sim.simxStartSimulation(clientID, sim.simx_opmode_oneshot)

#     # Main simulation loop
#     # while sim.simxGetConnectionId(clientID) != -1:
#     #     # Read joint positions
#     #     # errorCode, pos1 = sim.simxGetJointPosition(clientID,
joint1_handle, sim.simx_opmode_streaming)
#     #     # print(f"Joint1 position: {pos1}")
#     #     time.sleep(0.1)

#     # Stop simulation
#     # sim.simxStopSimulation(clientID, sim.simx_opmode_oneshot)

#     # Close the connection to CoppeliaSim
#     # sim.simxFinish(clientID)
# # else:
# #     print('Failed to connect to remote API server')
# print('Program ended')
```

Input

- **Simulation Software:** Robot model files, scene descriptions, physical parameters.
- **Python Scripts:** Desired joint angles, end-effector poses, forces, or commands for the simulated robot.

Expected Output

- **Simulation Software:** Visual representation of the robot moving according to kinematics and dynamics.
- **Python Scripts:** Logged data of joint trajectories, end-effector paths, sensor readings, and analysis results (e.g., plots of robot behavior).

Lab 5: Robot Vision with Raspberry Pi Camera

Title

Image Acquisition, Processing, and Object Detection with Raspberry Pi Camera

Aim

To perform image acquisition and processing using the Raspberry Pi camera and to implement basic object detection and recognition using Python libraries like OpenCV.

Procedure

1. **Setting up Raspberry Pi Camera:**
   - Connect the Raspberry Pi camera module to the CSI port.
   - Enable the camera interface in `raspi-config`.
   - Test camera functionality using `raspistill` or `raspivid` commands.
2. **Image Acquisition with Python:**
   - Install `picamera` library for Python.
   - Write a Python script to capture still images and video streams from the camera.
   - Save captured images to files.
3. **Image Processing with OpenCV:**
   - Install OpenCV library for Python (`opencv-python`).
   - Load an image into OpenCV.
   - Perform basic image processing operations:
     - Grayscale conversion.
     - Thresholding.
     - Blurring/smoothing.
     - Edge detection (e.g., Canny edge detector).
     - Color filtering (e.g., detecting objects within a specific color range using HSV).
4. **Basic Object Detection and Recognition:**
   - Use contour detection to find shapes in processed images.
   - Calculate properties of detected contours (area, perimeter, bounding box).
   - Implement simple object recognition based on shape, size, or color features.
   - (Optional) Use pre-trained Haar cascades for basic face or object detection.

Source Code

```
# Example Python script for image capture and basic processing with OpenCV
import cv2
from picamera2 import Picamera2 # For newer Raspberry Pi OS
import numpy as np
import time

# Initialize Picamera2
picam2 = Picamera2()
camera_config = picam2.create_preview_configuration(main={"size": (640,
480)})
picam2.configure(camera_config)
picam2.start()

time.sleep(2) # Allow camera to warm up

try:
    while True:
        # Capture frame
        frame = picam2.capture_array()
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB) # Convert to OpenCV's
BGR format

        # Convert to grayscale
        gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Apply Gaussian blur
        blurred_frame = cv2.GaussianBlur(gray_frame, (5, 5), 0)

        # Apply Canny edge detection
        edges = cv2.Canny(blurred_frame, 50, 150)

        # Display original and processed frames
        cv2.imshow("Original Frame", frame)
        cv2.imshow("Grayscale", gray_frame)
        cv2.imshow("Edges", edges)

        # Simple color detection (e.g., for a red object)
        # hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        # lower_red = np.array([0, 100, 100])
        # upper_red = np.array([10, 255, 255])
        # mask = cv2.inRange(hsv, lower_red, upper_red)
        # res = cv2.bitwise_and(frame, frame, mask=mask)
        # cv2.imshow("Red Object", res)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

except Exception as e:
    print(f"An error occurred: {e}")
finally:
    picam2.stop()
    cv2.destroyAllWindows()
    print("Camera stopped and windows closed.")
```

Input

- Physical objects in the camera's field of view.
- (Optional) User input for parameters like color ranges or threshold values.

Expected Output

- Live video feed displayed on the screen.
- Processed images showing grayscale, blurred, or edge-detected versions.
- Detected objects highlighted with bounding boxes or contours.
- Terminal output indicating detected objects or their properties.

Lab 6: Basic Object Recognition using Raspberry Pi

Title

Developing Python Programs for Camera Interfacing, Image Capture, and Basic Object Recognition

Aim

To develop Python programs for seamless camera interfacing, image capture, and to implement basic object recognition techniques using the Raspberry Pi. This lab builds upon the fundamentals of Lab 5.

Procedure

1. **Refined Camera Interfacing and Capture:**
   - Optimize image capture settings (resolution, frame rate, exposure) for specific recognition tasks.
   - Implement continuous video streaming for real-time processing.
   - Develop functions to save captured frames on demand or based on certain events.
2. **Advanced Image Pre-processing for Recognition:**
   - Explore different filtering techniques (e.g., median filter for noise reduction).
   - Implement image segmentation methods (e.g., Otsu's thresholding, watershed algorithm) to isolate objects of interest.
   - Perform morphological operations (erosion, dilation, opening, closing) to refine object shapes.
3. **Feature Extraction for Object Recognition:**
   - Extract features from segmented objects:
     - Geometric features: Area, perimeter, aspect ratio, circularity, solidity.
     - Color histograms.
     - Simple texture features.
   - Use these features to differentiate between different types of objects.
4. **Basic Object Classification:**
   - Develop a simple classification logic based on the extracted features (e.g., if aspect ratio is X and area is Y, then it's a square).
   - (Optional) Implement a basic K-Nearest Neighbors (KNN) classifier using `scikit-learn` for a small dataset of object features.
   - Display the recognized object's name or label on the video feed.

Source Code

```
# Example Python script for basic object recognition (conceptual)
import cv2
from picamera2 import Picamera2
import numpy as np
import time

picam2 = Picamera2()
camera_config = picam2.create_preview_configuration(main={"size": (640,
480)})
picam2.configure(camera_config)
picam2.start()
time.sleep(2)

def recognize_shape(contour):
    """
    Simple function to recognize basic shapes based on number of vertices.
    """
    approx = cv2.approxPolyDP(contour, 0.04 * cv2.arcLength(contour, True),
True)
    num_vertices = len(approx)
    shape = "unidentified"

    if num_vertices == 3:
        shape = "triangle"
    elif num_vertices == 4:
        # Check aspect ratio for square/rectangle
        x, y, w, h = cv2.boundingRect(approx)
        aspect_ratio = float(w) / h
        if 0.95 <= aspect_ratio <= 1.05:
            shape = "square"
        else:
            shape = "rectangle"
    elif num_vertices > 4:
        # For circles, check circularity or fit a circle
        area = cv2.contourArea(contour)
        (x, y), radius = cv2.minEnclosingCircle(contour)
        circle_area = np.pi * (radius ** 2)
        if area / circle_area > 0.8: # Simple circularity check
            shape = "circle"
        else:
            shape = "polygon"
    return shape

try:
    while True:
        frame = picam2.capture_array()
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        blurred = cv2.GaussianBlur(gray, (5, 5), 0)
        _, thresh = cv2.threshold(blurred, 60, 255, cv2.THRESH_BINARY_INV) #
Adjust threshold as needed

        contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

        for contour in contours:
            area = cv2.contourArea(contour)
            if area > 1000: # Filter small noise contours
                shape = recognize_shape(contour)
                x, y, w, h = cv2.boundingRect(contour)
                cv2.drawContours(frame, [contour], -1, (0, 255, 0), 2)
                cv2.putText(frame, shape, (x, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)

        cv2.imshow("Object Recognition", frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

except Exception as e:
    print(f"An error occurred: {e}")
finally:
    picam2.stop()
    cv2.destroyAllWindows()
    print("Camera stopped and windows closed.")
```

Input

- Physical objects with distinct shapes or colors placed in front of the camera.

Expected Output

- Live video feed with recognized objects labeled with their identified shape or type (e.g., "square", "circle", "triangle").
- Bounding boxes or contours drawn around detected objects.

Lab 7: Advanced Control System Implementation

Title

Design and Implementation of Advanced Control Systems (MPC) for Robot Control

Aim

To design and implement advanced control systems, specifically Model Predictive Control (MPC), for robot control tasks using Python, and to experiment with different control strategies and evaluate their performance.

Procedure

1. **Introduction to Advanced Control Concepts:**
   - Review limitations of traditional PID control.
   - Understand the principles of Model Predictive Control (MPC): prediction horizon, control horizon, cost function, constraints, optimization.
   - Explore other advanced control strategies (e.g., LQR, Sliding Mode Control) conceptually.
2. **System Modeling for MPC:**
   - Develop a mathematical model of a robot system (e.g., a mobile robot, a single joint of an arm) suitable for MPC. This typically involves state-space representation.
   - Linearize the model if necessary.
3. **Implementing MPC in Python:**
   - Choose a Python library for optimization (e.g., `scipy.optimize`, `CVXPY` for convex optimization, or specialized MPC libraries like `do-mpc`).
   - Formulate the MPC problem:
     - Define the prediction model.
     - Define the cost function (e.g., minimizing error, control effort).
     - Define constraints (e.g., motor limits, joint limits).
   - Implement the MPC algorithm to calculate optimal control inputs at each time step.
4. **Experimentation and Evaluation:**
   - Integrate the MPC controller with a simulated robot model (from Lab 4) or a real robot (if available and safe).
   - Test the MPC controller for various control tasks (e.g., trajectory tracking, obstacle avoidance, setpoint regulation).
   - Compare the performance of MPC with PID control in terms of accuracy, robustness, and handling of constraints.
   - Analyze the effects of changing MPC parameters (prediction horizon, control horizon, weights in the cost function).

Source Code

```
# Example Python conceptual structure for MPC (simplified)
# This would typically involve a more complex setup with an optimizer and
system model.
# from scipy.optimize import minimize
# import numpy as np

# class MPCController:
#     def __init__(self, model, cost_function, constraints, horizon):
#         self.model = model # Function that predicts future states
#         self.cost_function = cost_function # Function to minimize
#         self.constraints = constraints # Tuple of constraint functions
#         self.horizon = horizon # Prediction horizon

#     def calculate_control_input(self, current_state, setpoint):
#         # Define the optimization problem
#         # x0 = initial guess for control sequence
#         # bounds = limits for control inputs
#         # result = minimize(self.cost_function, x0, args=(current_state,
setpoint, self.model, self.horizon),
#         #                   method='SLSQP', bounds=bounds,
constraints=self.constraints)
#         # return result.x[0] # Return the first optimal control input
#         pass

# # Example usage (highly conceptual)
# # def robot_model(state, control_input):
# #     # Simulate robot dynamics for one time step
# #     # return next_state
# #     pass

# # def mpc_cost_function(control_sequence, current_state, setpoint, model,
horizon):
# #     # Calculate cost based on predicted trajectory and control effort
# #     # return total_cost
# #     pass

# # mpc = MPCController(model=robot_model, cost_function=mpc_cost_function,
constraints=None, horizon=10)
# # while True:
# #     current_state = read_robot_state()
# #     control_input = mpc.calculate_control_input(current_state,
target_setpoint)
# #     apply_control_input(control_input)
# #     time.sleep(dt)
```

Input

- Current state of the robot (e.g., position, velocity, orientation) from sensors.
- Desired setpoint or trajectory.
- System model parameters.

Expected Output

- The robot accurately tracks desired trajectories or reaches setpoints while respecting constraints.
- Performance metrics (e.g., tracking error, control effort, settling time) demonstrating the effectiveness of MPC.
- Comparison graphs or data showing the advantages of MPC over simpler control strategies.

Lab 8: Robot Navigation Simulation

Title

Robot Navigation Algorithm Development and Testing in ROS with Gazebo

Aim

To utilize simulation software, specifically ROS (Robot Operating System) with Gazebo, to develop and test robot navigation algorithms using Python, and to design and simulate various navigation scenarios with obstacles and dynamic environments.

Procedure

1. **Introduction to ROS and Gazebo for Navigation:**
   - Install ROS (e.g., Noetic or Humble) and Gazebo.
   - Understand ROS concepts: nodes, topics, services, messages, launch files.
   - Familiarize with Gazebo's interface for creating environments and spawning robots.
   - Learn about ROS navigation stack components (e.g., `amcl`, `move_base`, `gmapping`).
2. **Creating a Simulated Environment and Robot Model:**
   - Design a simple 2D or 3D environment in Gazebo with walls, obstacles, and a target destination.
   - Import or create a differential drive robot model (e.g., TurtleBot3) with appropriate sensors (Lidar, Odometry).
3. **Implementing Basic Navigation Algorithms in Python:**
   - **Odometry:** Subscribe to odometry topics to get robot pose.
   - **Lidar Processing:** Subscribe to Lidar scan topics and process raw scan data (e.g., find closest obstacle, detect free space).
   - **Simple Path Planning:** Implement a basic path planning algorithm (e.g., A* or Dijkstra's on a grid map, or a simple potential field method) in a Python ROS node.
   - **Velocity Control:** Publish velocity commands (`geometry_msgs/Twist`) to control the robot's movement.
4. **Designing and Simulating Navigation Scenarios:**
   - **Waypoint Navigation:** Program the robot to navigate to a series of predefined waypoints.
   - **Obstacle Avoidance:** Implement reactive obstacle avoidance logic using Lidar data.
   - **Dynamic Environments:** Introduce moving obstacles or changing environment layouts to test robustness.
   - Visualize robot movement, sensor readings, and planned paths in RViz (ROS visualization tool).

Source Code

```
# Example Python ROS Node for simple obstacle avoidance (conceptual)
# import rospy
# from geometry_msgs.msg import Twist
# from sensor_msgs.msg import LaserScan

# class ObstacleAvoider:
#     def __init__(self):
#         rospy.init_node('obstacle_avoider', anonymous=True)
#         self.cmd_vel_pub = rospy.Publisher('/cmd_vel', Twist, 
queue_size=10)
#         self.laser_sub = rospy.Subscriber('/scan', LaserScan, 
self.laser_callback)
#         self.twist = Twist()
#         self.forward_speed = 0.2 # m/s
#         self.turn_speed = 0.5 # rad/s
#         self.obstacle_distance_threshold = 0.5 # meters

#     def laser_callback(self, msg):
#         # Find the minimum distance in front of the robot
#         # This is a simplified example, usually involves processing a range 
of angles
#         front_ranges = msg.ranges[len(msg.ranges)//4 : 
len(msg.ranges)*3//4] # Example: front 180 degrees
#         min_distance = min(front_ranges) if front_ranges else float('inf')

#         if min_distance < self.obstacle_distance_threshold:
#             # Obstacle detected, turn
#             self.twist.linear.x = 0.0
#             self.twist.angular.z = self.turn_speed # Turn right (or left 
based on strategy)
#             rospy.loginfo("Obstacle detected! Turning.")
#         else:
#             # No obstacle, move forward
#             self.twist.linear.x = self.forward_speed
#             self.twist.angular.z = 0.0
#             rospy.loginfo("Path clear. Moving forward.")

#         self.cmd_vel_pub.publish(self.twist)

#     def run(self):
#         rospy.spin() # Keep the node running

# if __name__ == '__main__':
#     try:
#         avoider = ObstacleAvoider()
#         avoider.run()
#     except rospy.ROSInterruptException:
#         pass
```

Input

- **Gazebo:** World files (.world) and robot description files (.urdf/.xacro).
- **ROS:** Lidar scan data, odometry messages, goal poses published to `move_base`.
- **Python Scripts:** Target waypoints, navigation parameters.

Expected Output

- **Gazebo:** Robot moving autonomously within the simulated environment.
- **RViz:** Visualization of Lidar scans, robot pose, planned global and local paths, and obstacles.
- **Terminal:** ROS node output showing navigation status, obstacle detection, and control commands.
- The robot successfully navigates to target destinations while avoiding static and dynamic obstacles.

Lab 9: Autonomous Navigation

Title

Integrating Mapping, Localization, Path Planning, and Obstacle Avoidance for Autonomous Navigation

Aim

To integrate mapping, localization, path planning, and obstacle avoidance techniques to achieve full autonomous navigation for a robot in simulation, and to design and simulate various complex navigation scenarios.

Procedure

1. **Review and Setup:**
   - Ensure ROS and Gazebo are correctly configured from Lab 8.
   - Understand the full ROS Navigation Stack architecture (`tf`, `map_server`, `amcl`, `move_base`, `global_planner`, `local_planner`, `costmap_2d`).
2. **Mapping (SLAM - Simultaneous Localization and Mapping):**
   - Use `gmapping` or `cartographer` ROS packages to build a 2D occupancy grid map of an unknown environment by driving the robot around.
   - Save the generated map.
3. **Localization:**
   - Load the saved map using `map_server`.
   - Use `amcl` (Adaptive Monte Carlo Localization) to localize the robot within the known map using Lidar and odometry data.
   - Experiment with initial pose estimation and particle filter behavior.
4. **Global and Local Path Planning:**
   - Configure `move_base` with a global planner (e.g., A*, Dijkstra) and a local planner (e.g., DWA, TEB).
   - Understand how costmaps are generated and used by planners.
   - Send navigation goals to `move_base` (e.g., using RViz's "2D Nav Goal" tool or a Python script).
5. **Obstacle Avoidance and Dynamic Environments:**
   - Observe how the local planner and costmap handle dynamic obstacles (moving objects, people).
   - Tune costmap parameters to ensure safe and efficient obstacle avoidance.
   - Design and test scenarios with narrow passages, moving obstacles, and multiple goal points.
6. **Full Autonomous Navigation Implementation:**
   - Develop a Python script to automate sending navigation goals, monitoring navigation status, and handling potential recovery behaviors.
   - Test the integrated system in a complex simulated environment.

Source Code

```
# Example Python ROS Node for sending navigation goals (conceptual)
# This typically interacts with the 'move_base' action server.
# import rospy
# import actionlib
# from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
# from geometry_msgs.msg import PoseWithCovarianceStamped, Pose, Point,
Quaternion
```

```
# class Navigator:
#     def __init__(self):
#         rospy.init_node('simple_goal_sender', anonymous=True)
#         self.client = actionlib.SimpleActionClient('move_base',
MoveBaseAction)
#         rospy.loginfo("Waiting for move_base action server...")
#         self.client.wait_for_server()
#         rospy.loginfo("move_base action server found!")

#     def send_goal(self, x, y, yaw_degrees):
#         goal = MoveBaseGoal()
#         goal.target_pose.header.frame_id = "map"
#         goal.target_pose.header.stamp = rospy.Time.now()

#         goal.target_pose.pose.position.x = x
#         goal.target_pose.pose.position.y = y
#         # Convert yaw from degrees to quaternion
#         # from tf.transformations import quaternion_from_euler
#         # q = quaternion_from_euler(0, 0, math.radians(yaw_degrees))
#         # goal.target_pose.pose.orientation.x = q[0]
#         # goal.target_pose.pose.orientation.y = q[1]
#         # goal.target_pose.pose.orientation.z = q[2]
#         # goal.target_pose.pose.orientation.w = q[3]
#         goal.target_pose.pose.orientation.w = 1.0 # For simplicity, facing
forward

#         rospy.loginfo(f"Sending goal: ({x}, {y}) with yaw {yaw_degrees}
degrees")
#         self.client.send_goal(goal)
#         self.client.wait_for_result()
#         if self.client.get_state() == actionlib.GoalStatus.SUCCEEDED:
#             rospy.loginfo("Goal reached successfully!")
#             return True
#         else:
#             rospy.logwarn("Failed to reach goal.")
#             return False

# if __name__ == '__main__':
#     try:
#         navigator = Navigator()
#         # Example: Send robot to (1, 1) in map frame with 0 yaw
#         # navigator.send_goal(1.0, 1.0, 0.0)
#         # time.sleep(2)
#         # navigator.send_goal(2.0, 0.0, 90.0)
#         navigator.send_goal(0.0, 0.0, 0.0) # Return to origin
#     except rospy.ROSInterruptException:
#         pass
```

Input

- **ROS:** Robot motion commands (teleop), initial pose estimate for `amcl`, navigation goals from RViz or Python scripts.
- **Gazebo:** Environment models, robot models.

Expected Output

- **RViz:** Real-time visualization of the generated map, robot's estimated pose, Lidar scans, global path, local path, and costmaps.
- **Gazebo:** Robot moving autonomously, avoiding obstacles, and reaching target destinations.
- **Terminal:** ROS node messages indicating mapping progress, localization accuracy, path planning status, and goal achievement.

- The robot successfully navigates complex environments autonomously, demonstrating robust mapping, localization, and navigation capabilities.

Lab 10: Machine Learning for Robot Control

Title

Training and Implementing Machine Learning Models for Robot Control Tasks

Aim

To train and implement machine learning models using Python for specific robot control tasks (e.g., object recognition, path planning), and to evaluate the performance of these models and refine them for improved results.

Procedure

1. **Introduction to Machine Learning in Robotics:**
    - Review fundamental ML concepts: supervised learning, unsupervised learning, reinforcement learning.
    - Understand common ML algorithms applicable to robotics (e.g., neural networks, support vector machines, decision trees, clustering).
    - Discuss applications: object detection, semantic segmentation, path planning, grasping, human-robot interaction.
2. **Data Collection and Preprocessing:**
    - For a chosen task (e.g., recognizing different types of objects, classifying traversable terrain):
        - Collect a dataset of sensor readings (images, Lidar scans, joint states) and corresponding labels or desired actions.
        - Preprocess the data (e.g., resizing images, normalization, feature scaling).
3. **Model Training:**
    - Choose an appropriate ML framework (e.g., TensorFlow, Keras, PyTorch, scikit-learn).
    - Select an ML model architecture suitable for the task (e.g., a simple CNN for image classification, a feedforward neural network for state-action mapping).
    - Train the model using the collected dataset.
    - Monitor training progress (loss, accuracy).
4. **Model Implementation and Deployment:**
    - Integrate the trained ML model into a Python script running on the Raspberry Pi or in a simulation environment.
    - For object recognition: Use the model to classify objects detected by the camera.
    - For path planning: Use the model to predict optimal actions based on sensor input and current state.
5. **Evaluation and Refinement:**
    - Evaluate the model's performance using metrics relevant to the task (e.g., accuracy, precision, recall for classification; tracking error for control).
    - Identify areas for improvement (e.g., collect more data, adjust model architecture, fine-tune hyperparameters).
    - Iteratively refine the model and re-evaluate.

Source Code

```
# Example Python script for a simple image classification with a pre-trained
model (conceptual)
# import cv2
# from picamera2 import Picamera2
# import numpy as np
# from tensorflow.keras.applications.mobilenet_v2 import preprocess_input,
decode_predictions
# from tensorflow.keras.models import MobileNetV2
# import time

# picam2 = Picamera2()
# camera_config = picam2.create_preview_configuration(main={"size": (224,
224)}) # MobileNetV2 input size
# picam2.configure(camera_config)
# picam2.start()
# time.sleep(2)

# # Load pre-trained MobileNetV2 model
# model = MobileNetV2(weights='imagenet')

# try:
#     while True:
#         frame = picam2.capture_array()
#         frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
#         frame_resized = cv2.resize(frame, (224, 224))
#         image_array = np.expand_dims(frame_resized, axis=0)
#         preprocessed_image = preprocess_input(image_array)

#         predictions = model.predict(preprocessed_image)
#         decoded_predictions = decode_predictions(predictions, top=3)[0]

#         label = f"{decoded_predictions[0][1]}:
{decoded_predictions[0][2]*100:.2f}%"
#         cv2.putText(frame, label, (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 1,
(0, 255, 0), 2, cv2.LINE_AA)

#         cv2.imshow("ML Object Recognition", frame)

#         if cv2.waitKey(1) & 0xFF == ord('q'):
#             break

# except Exception as e:
#     print(f"An error occurred: {e}")
# finally:
#     picam2.stop()
#     cv2.destroyAllWindows()
#     print("Camera stopped and windows closed.")
```

Input

- **Training:** Labeled datasets (images, sensor readings, etc.).
- **Inference:** Real-time sensor data from the robot (camera feed, Lidar scans, joint states).

Expected Output

- **Training:** Trained model file, training loss/accuracy curves.
- **Inference:**
  - For object recognition: Live video feed with recognized objects labeled and confidence scores.
  - For path planning/control: Robot executing actions predicted by the ML model, demonstrating intelligent behavior (e.g., navigating complex environments, performing specific manipulation tasks).
- Performance metrics validating the model's effectiveness.

Lab 13, 14, 15: Robotics Project

Title

Robotics Project: Design, Implementation, and Documentation

Aim

To apply the knowledge and skills acquired throughout the program to design, implement, test, and document a comprehensive robotics or automation project, while also considering relevant ethical implications.

Procedure

1. **Project Scope and Objectives Definition (Lab 13):**
    - Identify a problem or a specific task in robotics/automation.
    - Clearly define the project's scope, goals, and success criteria.
    - Break down the project into manageable sub-tasks.
    - Conduct a literature review and research on the chosen topic.
    - Identify and consider potential ethical implications related to the project (e.g., data privacy, safety, bias in AI).
2. **Design and Implementation (Lab 15):**
    - Develop a detailed design for the robotic system or automation solution, including hardware components (if applicable), software architecture, and algorithms.
    - Implement the project using Python, Raspberry Pi, and other suitable tools and technologies (e.g., ROS, OpenCV, TensorFlow, specific sensors/actuators).
    - Modularize the code and ensure good programming practices.
3. **Testing and Evaluation (Lab 15):**
    - Develop a testing plan to verify the functionality and performance of the project.
    - Conduct rigorous testing in simulation and/or on physical hardware.
    - Collect data to evaluate the project's performance against the defined objectives.
    - Address any ethical concerns identified during the research phase, potentially through design choices or mitigation strategies.
4. **Documentation and Presentation (Lab 15):**
    - Prepare comprehensive documentation of the project, including:
        - Project proposal.
        - Design document.
        - Implementation details (code comments, architecture diagrams).
        - Testing results and analysis.
        - Discussion of ethical considerations and how they were addressed.
        - Conclusion and future work.
    - Prepare a presentation to showcase the project's functionality, design, challenges, and outcomes.

Source Code

- **This section will contain the complete source code for your specific project.**
- It should be well-commented, organized into logical modules/files, and include any necessary configuration files.
- Example:
- `# main_robot_control.py`
- `# This file would integrate all components of your final project.`

```
# import sensor_module
# import control_module
# import vision_module
# import navigation_module

# if __name__ == "__main__":
#     # Initialize modules
#     # Start main loop
#     # Handle sensor data, process it, make decisions, control robot
#     pass
```

Input

- **This section will list all inputs required for your specific project.**
- Examples: Sensor data (camera, Lidar, IMU), user commands, predefined maps, configuration parameters, target coordinates.

Expected Output

- **This section will describe the expected outcomes and behaviors of your specific project.**
- Examples:
    - A mobile robot autonomously navigating a complex environment.
    - A robotic arm successfully picking and placing objects.
    - An automated system performing a specific industrial task.
    - A clear demonstration of the project's objectives being met.
    - Detailed performance metrics and analysis from testing.
    - A well-structured project report and a compelling presentation.