Ahmedabad University

ECE 504 - Internet of Things

Final Report

Section Number - 1

Submitted to faculty: Prof. Anurag Lakhlani

Date of Submission: 19/11/2024

Student Details

Roll No.	Name of the Student	Name of the Program
AU2140070	Dhruv Jagani	BTech Mechanical
AU2140065	Kanad Patel	BTech Mechanical
AU2140115	Nishant Patel	BTech Mechanical
AU2140096	Ansh Virani	BTech Mechanical

2021-2022 (Monsoon Semester)

Chapter 1: Introduction

We aim to develop a scalable system that utilizes omnidirectional robots designed for warehouse inventory management. These robots will communicate with the central system using wireless protocols, and the primary central system will have a live camera feed to identify ArUco markers so they can navigate efficiently through the warehouse. Implementing IoT devices like Raspberry Pi and ESP32 modules will allow real-time data processing and communication. The robots are designed to handle heavy tasks like relocating boxes and optimizing operations in large warehouses where human movement in and about the warehouse leads to inefficient utilization of time and resources by the warehouse workers. Moreover, the more movement around the warehouse, the more the workers will be prone to injuries. By equipping them with omnidirectional wheels, we aim to enhance mobility while minimizing the weight of control systems. This will result in high-functioning, efficient robots with increased operational versatility.

1.1 Learning Goals

Through this project, we aim to learn and get an in-depth understanding of the following critical components of the Internet of Things:

1. **IoT Communication Systems:** Wireless communication protocols and technologies enable seamless communication between the robots, allowing robots and a centralized control system to exchange data and instructions quickly. This will utilize technologies such as Raspberry Pi as the centralized control and ESP32 module as the local brains of each robot to enable real-time control and monitoring within an Internet of Things network.
2. **Edge Computing in IoT:** Understanding how edge devices, such as robots with onboard sensors and microcontrollers, process data locally and make decisions in real-time before sending only the most essential information back to the central system is known as edge computing in the Internet of Things. This will enable us to investigate higher efficiency and lower latency advantages in IoT-based systems to create a more robust and reliable system.

3. **Navigation and Localization Algorithms:** Learning how ArUco markers and camera-based navigation systems are combined with IoT devices to enable autonomous navigation in a warehouse setting with high accuracy. We want to investigate how real-time sensor feedback and data analysis might help IoT-connected devices improve the efficiency of pathfinding and object recognition.
4. **Collaborative IoT Systems:** Research on how several IoT-capable robots might work together and exchange information in a shared space to do tasks on their own. We will investigate methods to improve the efficiency of IoT-enabled robotic systems, such as decentralized decision-making, resource sharing, and synchronized work distribution.

1.2 Project Utility

This project will streamline warehouse operations by automating repetitive and physically demanding tasks. The robots will serve as reliable, precise, and efficient warehouse workers, particularly those handling heavy loads or large volumes of inventory, reducing the potential for human errors and injuries and increasing operational efficiency.

Chapter 2: Literature Review

Autonomous mobile robots have emerged as essential tools in modern warehouse management, optimizing operations and reducing human injuries. In our project, we leverage ArUco markers for navigation, IoT technologies for real-time communication, and omnidirectional wheels to improve mobility and flexibility in confined spaces to achieve our goals. This chapter explores the technological foundation and research that informs the development of our robots.

2.1 ArUco Markers for Navigation

ArUco markers are a crucial technology we integrate with our project for locating and navigation. ArUco markers, which offer excellent precision in indoor and outdoor contexts, are binary square fiducial markers used to estimate the robot's position within a specific space. Their usefulness in several applications, such as autonomous navigation and object tracking, has been demonstrated by research in this field.

The OpenCV documentation, which offers guidance on identifying and decoding ArUco markers with the OpenCV library, is a vital source of information on this subject. Because of this library's efficiency and versatility, computer vision applications use it widely. Because ArUco markers can be detected and interpreted with little computing overhead, they are perfect for systems with limited resources, such as our project's Raspberry Pi platform.

Moreover, the possibility of these markers for accurate indoor navigation is investigated in the paper "Indoor Navigation of an Autonomous Guided Vehicle Using ArUco Markers." This study demonstrates that ArUco markers, as opposed to conventional infrared or ultrasonic sensors, can significantly increase localization accuracy. A reliable approach for warehouse environments where barriers and tight lanes could interfere with standard sensing technologies is to combine ArUco markers with a camera-based system.

There's solid research supporting this approach of using ArUco markers for navigation in conjunction with camera based centralized systems, and hence we have decided to move forward with the approach, while trying to incorporate our own ideas and innovations to the project.

2.2 IoT-Driven Communication and Control

IoT technologies, including ESP modules and Raspberry Pi, are the foundation of our project's communication system. IoT makes a smooth connection between the robots and the central system possible. This is essential for data transmission, task assignment, and real-time monitoring—all vital to the autonomous system's effectiveness.

The application of IoT to industrial automation is covered in a substantial body of research. Kumar et al. (2018) examined how the Internet of Things affects control systems and industrial automation. They highlighted how IoT devices could improve industrial decision-making processes by integrating various systems to deliver real-time data. This is particularly important for warehouse management since accuracy and efficiency are crucial.

Our project uses a Raspberry Pi and ESP32 module combo to enable communication between the robots and the central system. The main controller of the project is a Raspberry Pi with a live camera feed, and the ESP32 controls the robot’s sensors and motors. Both the Raspberry Pi and ESP32 are IoT enabled devices, making them idle for wireless communication. In a warehouse environment, this multi-tiered system guarantees that the robots are dependable and responsive.

2.3 Omni-Directional Wheels and Mobility

For warehouse robots, mobility is significant, especially in small areas. Unlike conventional wheeled robots, ours can travel in any direction because of the omnidirectional wheels, which give them greater agility. This adaptability is necessary to maneuver through tight corners, packed spaces, and narrow warehouse aisles. These wheels enable the robot to be holonomic.

Gross et al. (2006) explored the benefits of omnidirectional wheels in autonomous systems as part of their study on mobile robotics. According to their research, these wheels allowed robots to move more precisely and with more excellent skill, which made them perfect for jobs like selecting and placing in warehouses. Omnidirectional wheels also simplify the control system and reduce the robot's overall weight by reducing the mechanical components needed for steering.

2.4 Warehouse Automation and Error Reduction

As the e-commerce and logistics sectors grow, warehouse automation is becoming an essential field of study. Automated systems that use autonomous robots can significantly increase productivity by reducing the need for human interaction in repetitive operations like choosing and arranging objects which are hectic. Amazon purchased the Kiva System, an example of this kind of automation. Autonomous robots transfer merchandise/objects to workers instead of having employees fetch products themselves. This method expedites warehousing processes, reduces human error, and also reduces their daily workloads.

Azadeh et al. (2017) examined the advantages of employing autonomous robots in sizable warehouses in their study on warehouse automation. According to their research, robots may reduce error rates by 15% and boost production by 25%. These results align with our project's goals since our robots will help to reduce human error while increasing operational effectiveness in the warehouse.

Moreover, Lee and Bagheri's (2015) assessment of the state of industrial automation trends noted that combining autonomous robots with IoT systems might significantly improve the scalability of warehouse operations. Businesses can decrease the possibility of lost items and increase tracking accuracy by integrating robots with a centralized inventory system (Aruco).

2.5 Path Optimization and Task Scheduling

Path optimization is yet another essential part of our endeavor. Accurate localization is simply one factor in a robot's ability to navigate a warehouse; other essential factors are task scheduling and path planning. Algorithms that optimize robot mobility to reduce travel time and energy consumption have been the subject of numerous studies.

For example, the multi-robot route planning research by Masehian and Sedighizadeh (2010) offers a thorough summary of the techniques that can be used to maximize robot mobility in a shared environment. Their research is especially pertinent to our proposal, which calls for the cooperative usage of several robots in an Arco warehouse. Effective path planning minimizes the chance of collisions and guarantees task completion promptly.

2.6 Market Survey of Current Products

Several companies and research institutions have developed robotics systems to improve warehouse management. Here, we compare three essential products:

1. **Kiva Systems (Amazon Robotics)**

   Overview: Amazon Robotics (previously known as Kiva Systems) uses autonomous robots in their fulfillment centers. These robots carry entire shelves of items to human workers, allowing them to retrieve items quickly.

   Key Features:

   - Uses a grid-based navigation system.
   - Equipped with sensors to avoid collisions.
   - Fully integrated with Amazon's inventory management system.

   Pros:

   - High speed and efficiency.
   - Significantly reduces the walking time for workers.
   - Fully automated fulfillment process.

   Cons:

   - Expensive to implement and maintain.
   - Requires a specialized warehouse layout.

2. **Fetch Robotics**

   Overview: Fetch Robotics offers collaborative robots designed for material handling, specifically for pick-and-place and transportation tasks.

   Key Features:

   - Robots use cloud-based software to navigate and carry out tasks.
   - Easily integrate into existing warehouses without needing significant layout changes.

   Pros:

   - Adaptable to different warehouse environments.
   - Quick implementation due to easy integration.
   - Capable of collaborating with human workers.

   Cons:

   - Limited load-carrying capacity compared to other robots.

3. **GreyOrange Robotics**

   Overview: GreyOrange offers automated robotic systems to automate the entire warehousing process, from storage to packing.

   Key Features:

   - Uses AI to optimize inventory placement and retrieval.
   - It is equipped with advanced sensors and real-time communication systems.
   - Can handle various tasks such as sorting, picking, and packing.

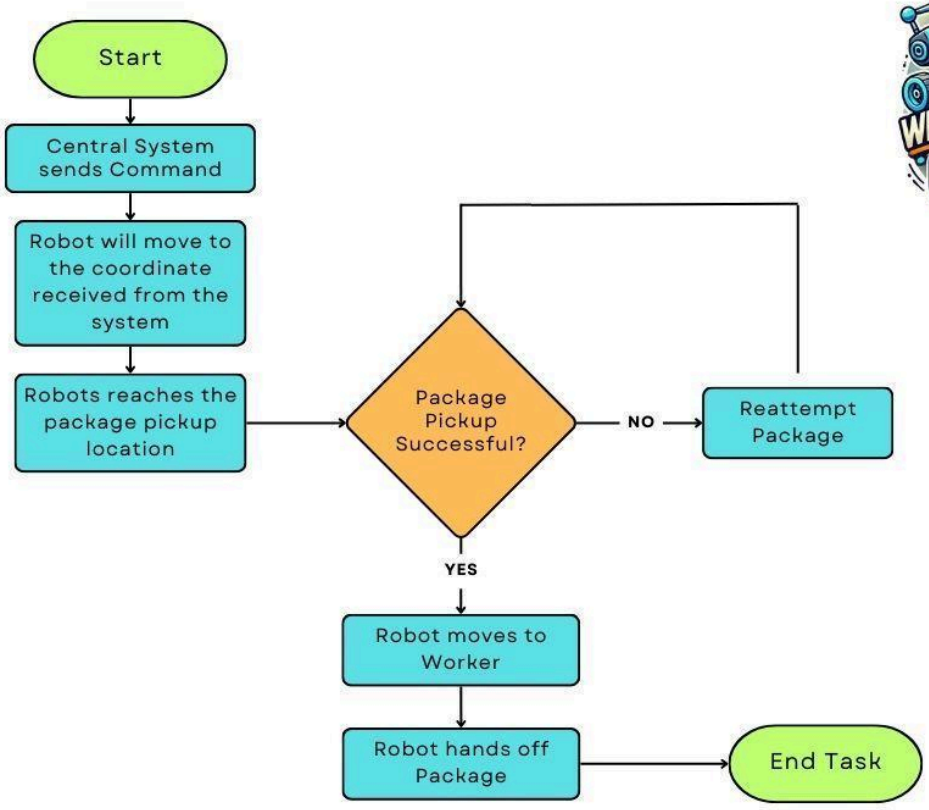
     Pros:

   - AI-powered systems optimize warehouse operations.
   - Scalable solution for large warehouses.

     Cons:

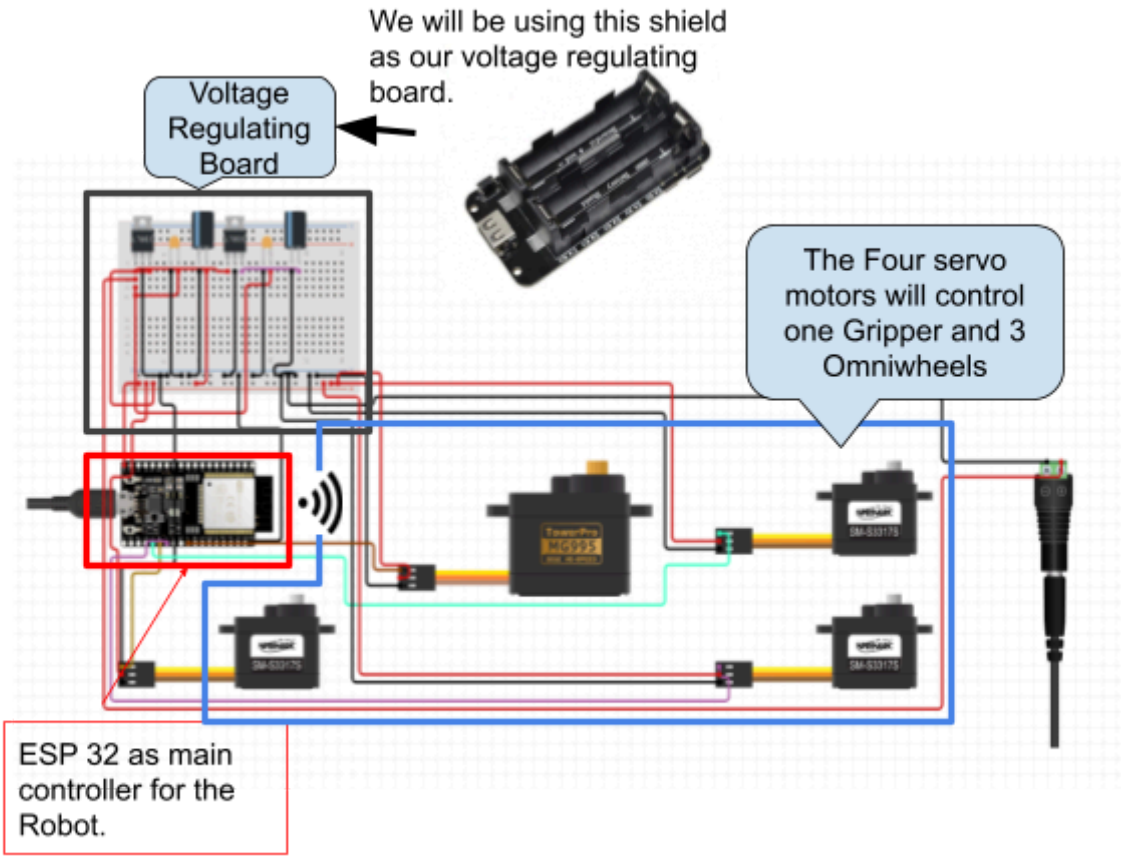
   - High initial setup cost.

Chapter 3: Block Diagram

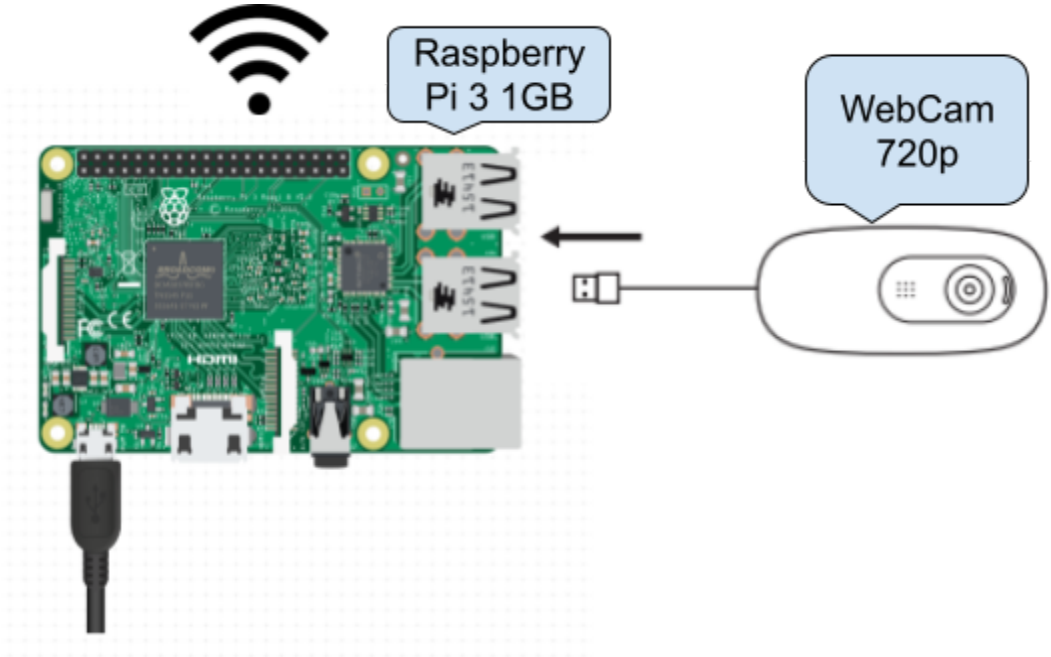


The aforementioned flow chart explains the brief idea that our team has with respect to the project, once the system is enabled the robots will receive coordinates from the centralized system. The robot will start moving towards those coordinates and after successfully reaching that location, will try to pick up the package. After successfully picking up the package, it will start moving towards the worker's location and hand off the package. At the end, the robot will wait for further instructions repeating this cycle many times till the entire order is fulfilled.

Chapter 4: Circuit Diagram



Circuit Diagram for the Robot



Circuit Diagram for the main Controller

Technical Inputs and Outputs of the Project

Inputs

Visual Data from Aruco Markers:

- **Sensor:** Logitech C270 HD Webcam.
- **Rationale:** The webcam will provide a good enough real-time fill-in video for detecting Aruco markers in the environment. Moreover, it is inexpensive and very compatible with OpenCV and ROS 2 for marker detecting. The frame rate and field of view should be enough to monitor robots in small to medium-sized warehouse areas.
- **Input Functionality:** It will provide the necessary video input for finding the location and orientation of both robots and objects, using Aruco markers in the warehouse.

Robot Localization Data:

- **Sensors:** Aruco markers and OpenCV for its detection.
- **Justification of Choice:** ArUco markers are simple, lightweight, and quite easy to implement when performing indoor localization. They are characterized by high accuracy of pose estimation, meaning position and orientation, with very low computational overhead. This means they are able to perform well in environments where very accurate localization is important.
- **Input Role:** This would be employed in robot navigation whereby each of them is allowed to determine its position concerning key points in the warehouse, including the home position of the object, the location it is going to set an object among others.

Robot Control Commands:

- **Source:** Centralized controller running on a Raspberry Pi.
- **Selection Reason:** ROS2 has powerful communication tools and real-time control. The centralized system will be able to simultaneously operate several robots and send precise navigation goals.
- **Input Role:** This would be the commands generated from the Raspberry Pi based on visual data and sent to the ESP32 microcontrollers on each robot to execute.

Data for Wi-Fi Communication:

- **Modules:** ESP32 Wi-Fi module.
- **Rationale for Choice:** ESP32 has on board Wi-Fi and also can run Micro-ROS which will seamlessly communicate with the Raspberry Pi. Also, it is low-cost, highly efficient in handling lightweight communication in real time.

- **Input Functionality:** Allows for wireless communication between the central controller and mobile robots regarding the sending of task commands and feedback.

Outputs

Robot Navigation:

- **Actuators:** Servo Motors (TowerPro MG995).
- **Selection Reason:** Chosen because these servo motors provide an excellent amount of precision and fine movement control, which becomes crucial in the case of omni-wheeled robots. They provide enough torque to move the robot at a precisely regulated speed.
- **Output Role:** The movement of the omni-wheeled robots in conformation with the navigation provided through the ESP32 microcontroller will depend on them.

Object Manipulation:

- **Actuators:** Gripper controlled by servo motors.
- **Reason for Selection:** Servos allow for precise actuation/opening/closing of the gripper. Selected servo motors, such as TowerPro MG995, provide adequate torque that would ensure proper operation of the gripper.
- **Output Role:** Picks up and drops objects from the warehouse upon command.

Feedback of Task Completion:

- **Powered by the ESP32 microcontroller.**
- **Reason for Selection:** ESP32 is capable of communicating the task status back to the central controller over Wi-Fi and thus provides the capability for real-time monitoring of the progress of robots. Lightweight architecture allows for fast feedback loops.
- **Output Role:** Feedback on status of completion, like object picked, reached end position, or task failed, to the Raspberry Pi for further task allocation.

Collision Avoidance Actions:

- **Sensor Input:** Visual detection via Aruco markers and camera.
- **Reason for Selection:** The visual system can detect the object or another robot in proximity to it in real time and feed this information into collision avoidance algorithms to make real-time adjustments in navigation.
- **Output Role:** Changing the path of movement of the robot, based on the processing of visual information by the Raspberry Pi, to avoid collision will make the operation of the robot safe and efficient in an environment where space is shared.

Sensor and Component Selection Justification:

Aruco Markers for Localization:

- **Why Aruco?** Aruco markers are an efficient way to have precise pose estimation in low computational resources and at a low cost. Hence, they are highly preferred in indoor warehouse settings where precision in localization is the key for efficient navigation of robots.

Logitech C270 Webcam:

- **Why This Camera?** Logitech C270 offers adequate resolution of 720p and appropriate frame rate for real-time Aruco marker detection. It easily integrates with OpenCV, hence making it work seamlessly with ROS2 for visual localization-related tasks.

ESP32 Microcontrollers:

- **Why ESP32?** The reason behind using the ESP32 is that it is cost-effective; it has built-in Wi-Fi for communication, and it also supports Micro-ROS, which will enable real-time control of the robots. The ESP32 is also light and power-efficient, hence suitable for mobile robots.

Omni-Wheeled Robots:

- **Why Omni-Wheels?** Omni-wheels will allow for multidirectional movement and thus high agility when it comes to navigating through tight spaces. This becomes necessary in warehouse environments, where flexibility and precision may be in demand for object collection and transportation.

Servo Motors TowerPro MG995:

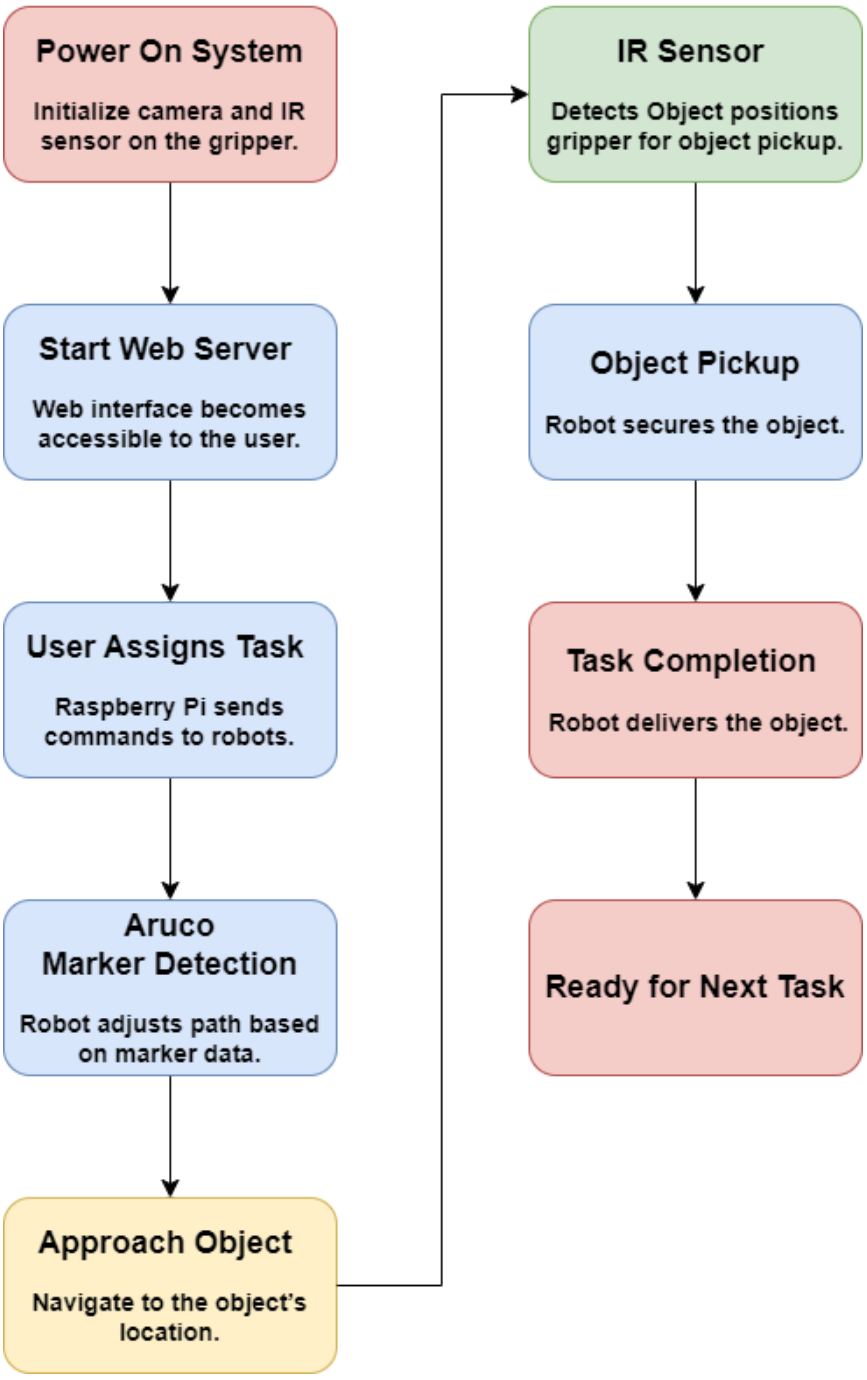
- **Why These Servos?** These servo motors provide reliable torque and speed control useful for both movement and gripper operation, which ensures smooth task execution and accurate positioning while handling objects.

Chapter 5: Arduino UNO/Mega OR Raspberry Pi Features

Platform	Processor	Clock Speed	RAM	Storage	GPIO Pins
Arduino	8-bit AVR or ARM	16 MHz (Uno)	2 KB (Uno)	Flash memory (32 KB)	14 (Uno)
Raspberry Pi	ARM Cortex-A53	1.5 GHz	2 GB (Pi 4)	MicroSD	40
BeagleBone	AM335x 1GHz ARM Cortex-A8	1 GHz	512 MB	4GB eMMC	65
Intel Galileo	Quark SoC X1000	400 MHz	256 MB	MicroSD	14
Intel Edison	Dual-core Intel Atom + Quark	500 MHz	1 GB	4GB eMMC + microSD	40

Platform	Operating System	Connectivity	Power Consumption	Best For	Strengths	Weaknesses
Arduino	None	USB	Serial	Low	Basic embedded projects	Limited processing power
Raspberry Pi	Linux (Raspbian)	Wi-Fi	Ethernet	Bluetooth	Moderate	Multimedia and IoT projects
BeagleBone	Linux (Debian)	Wi-Fi	Ethernet	USB	Moderate	Industrial and embedded systems
Intel Galileo	Linux (Yocto)	Ethernet	USB	Moderate	IoT and prototyping	Intel Architecture
Intel Edison	Linux (Yocto)	Wi-Fi	Bluetooth	Moderate	Wearable and IoT devices	Small form factor

Chapter 6: Program Flow Chart



Chapter 7: Sensors

1. Infrared (IR) Sensor: TCRT5000

Description: The TCRT5000 IR Sensor is a reflective optical sensor that detects the presence of objects or surfaces. It includes an infrared emitter and a phototransistor that detects the reflected IR light.
Key Specifications:

- **Operating Voltage:** 3.3V to 5V
- **Detection Range:** 1mm to 25mm
- **Response Time:** 1ms
- **Dimensions:** 30mm x 15mm x 12mm

Operating Principle:

The IR sensor emits infrared light from the emitter. When an object is placed in front of the sensor, the light reflects back to the phototransistor. The amount of reflected light depends on the object's distance and reflectivity, allowing the sensor to detect the presence of objects or follow a line on the ground.

Application in the Project:

- **Line Following:** The sensor can be used to implement line-following mechanisms for mobile robots in the warehouse. The robot will follow pre-set paths marked by reflective tape on the ground, assisting with navigation.
- **Edge Detection:** It will help the robot avoid falling off ledges by detecting changes in surface reflections.

Pin Description:

- **VCC:** 3.3V or 5V Power
- **GND:** Ground
- **OUT:** Digital output (HIGH when no object is detected; LOW when an object is detected)

2. Camera (For Aruco Marker Detection)

Description: A Logitech C270 HD Webcam or a similar camera is used for detecting Aruco markers, enabling precise localization and navigation of mobile robots within the warehouse. The camera captures visual input and sends it to the Raspberry Pi for processing.
Key Specifications:

- **Resolution:** 720p (1280 x 720)
- **Field of View:** 60° (Diagonal)
- **Frame Rate:** 30fps
- **Connection:** USB 2.0
- **Dimensions:** 29mm x 72mm x 62mm

Operating Principle:

The camera captures images of the environment and processes them using OpenCV on the Raspberry Pi to detect Aruco markers. Each marker has a unique ID, which allows the robot to determine its precise position and orientation within the warehouse. The system uses this information to guide the robots through various tasks.

Applications in Your Project:

- **Localization:** Aruco markers placed throughout the warehouse allow robots to determine their exact position and orientation.
- **Object Identification:** Markers attached to objects enable the robots to identify and move toward specific items for pickup or transport.

Connection:

The camera connects via USB to the Raspberry Pi and works seamlessly with OpenCV libraries for image processing.

Chapter 8: Actuators & Display

1. Actuator: Servo Motor TowerPro MG995

Description: The TowerPro MG995 is an off-the-shelf servo motor applicable in many robotic and automation projects. This servo has high torque, which makes it suitable for driving mechanical systems with a high load. It is controlled by a Pulse Width Modulation (PWM) signal and can rotate its shaft up to a particular position depending on the PWM duty cycle.
Key Specifications:

- **Operating Voltage:** 4.8V - 6.6V DC
- **Torque:** 10 kg/cm at 4.8V; 12kg/cm at 6V
- **Speed:** 0.2 sec/60° at 4.8V; 0.16 sec/60° at 6V
- **Weight:** 55g
- **Dimensions:** 40.7mm × 19.7mm × 42.9mm
- **Control:** PWM (Pulse Width Modulation)
- **Rotation Range:** 180° (from the neutral point, 90° on either side)
- **Connector:** 3-pin connector for Power, Ground, and Signal

Operating Principle:

The servo motor is controlled by a PWM signal. The angle of the motor shaft is dependent on the time of the high pulse in a repeating signal cycle. Normally, a pulse of 1ms corresponds to 0°, 1.5ms to 90° (neutral), and 2ms to 180°. The motor receives these signals from a microcontroller (in this case, the Raspberry Pi via its GPIO pin) and adjusts its position accordingly.

Applications in the Project:

The MG995 servo motor will drive physical movements like rotating or positioning an attached gripper or mechanical component, responding to commands from the Raspberry Pi via the web interface.

Pin Description:

- **Red Wire:** Power (4.8V to 6V)
- **Brown Wire:** Ground (GND)
- **Orange Wire:** PWM Signal (to control rotation)

2. Display Device: Laptop Screen Interface

Description:
The project uses a laptop screen as the central interface for monitoring and controlling the system. Instead of relying on a dedicated hardware display or a Raspberry Pi-hosted server, the laptop hosts a real-time web-based interface for interacting with the actuator (servo motor). This setup provides a simple, accessible, and dynamic control system, leveraging the laptop's browser and processing capabilities.

Key Features:

- **Dynamic Interface**: A web interface (served from the laptop using Flask or a similar framework) allows users to monitor and control the servo motor in real-time.
- **Local Access**: The control panel is directly accessible on the laptop's browser, eliminating the need for external hosting or additional devices.
- **Customizable Controls**: The interface features interactive elements such as sliders, buttons, and input fields to set the servo motor's position and display real-time feedback.
- **No Additional Hardware**: The laptop serves as the control hub, reducing the need for external display devices like LCDs or additional hardware.

Operating Principle:
A Python-based web server hosted on the laptop listens for user input through the web interface. Users interact with controls on the webpage, sending commands to adjust the servo motor's position. The laptop processes these commands and generates corresponding PWM signals for the servo motor. The motor's current position can also be displayed in real-time on the same interface.

Applications in the Project:

- Users can control the servo motor's angle through the laptop's browser interface, making the system portable and easy to set up.
- Real-time feedback, such as the motor's current position, is displayed directly on the laptop screen for immediate monitoring and adjustment.

Final Setup Overview:

- **Actuator (Servo Motor):** The MG995 servo motor drives the physical movement of the system (e.g., robotic arm or gripper) based on user input from the web interface.
- **Display System (Web Page Server):** The web interface replaces traditional display methods, providing real-time control and feedback.

- **Control via Web Server:** The flask on the Raspberry Pi handles user inputs and sends control signals to the servo motor. This approach eliminates the need for a physical LCD and allows remote, flexible access.

Chapter 9: Working Video

Chapter 10: Details of Communication Protocols

Wi-Fi:

- The ESP32 connects to a laptop or Raspberry Pi via a Wi-Fi network (hotspot or router).
- Communication between devices occurs through HTTP requests (RESTful API) or WebSocket for real-time updates.
- This enables seamless wireless data transfer for commands (e.g., servo angle) and feedback (e.g., robot position).

HTTP:

- The Flask server running on the laptop sends and receives data to/from the ESP32 via HTTP GET/POST requests.
- This protocol is simple and sufficient for non-time-critical tasks like setting servo angles.

Protocol	**Range**	**Speed**	**Power Consumption**	**Complexity**	**Best Use Case**
Wi-Fi	~50-100 meters indoors	Up to 600 Mbps (802.11n)	Moderate to High	Moderate	High-speed data transfer over moderate distances.
Bluetooth	~10-30 meters	Up to 2 Mbps (BLE 5.0)	Low	Simple	Short-range, low-power IoT devices.
Zigbee	~10-100 meters	~250 Kbps	Very Low	High (requires Zigbee stack)	Low-power mesh networks for IoT (e.g., sensors).
LoRaWAN	~10 km (urban) to ~15 km (rural)	~0.3-50 Kbps	Very Low	High	Long-range, low-data IoT communication.
NFC	~10 cm	~424 Kbps	Very Low	Simple	Contactless communication (e.g., payments).
Infrared (IR)	Line-of-sight, ~10 meters	~4 Mbps	Low	Simple	Remote control systems.
Cellular (4G/5G)	Up to several kilometers	100 Mbps (4G), 10 Gbps (5G)	High	High	High-speed, long-range communication.

Wi-Fi

Overview: Wi-Fi is a wireless networking technology that uses radio waves to provide high-speed internet and network connections. It is widely used in IoT applications for its reliability, speed, and ease of integration.

Key Features:

- High-speed data transfer rates (up to 600 Mbps with 802.11n).
- Moderate to high range (~50-100 meters indoors).
- Supports a wide variety of devices and network architectures.

Advantages:

- High bandwidth supports simultaneous communication with multiple devices.
- Easy integration with existing infrastructure (routers, hotspots, etc.).
- Secure communication with WPA2/WPA3 encryption.

Disadvantages:

- Relatively high power consumption.
- Requires a pre-existing Wi-Fi network or hotspot.

Applications in the Project:

- Real-time data exchange between the ESP32 and the laptop via HTTP and WebSocket protocols.
- Hosting the control interface (when using Raspberry Pi) or directly communicating with the laptop.

Bluetooth

Overview: Bluetooth is a short-range wireless communication technology designed for low-power IoT applications. It is commonly used in personal devices and low-energy communication systems.

Key Features:

- Short-range (~10-30 meters).
- Low power consumption, particularly with Bluetooth Low Energy (BLE).
- Simple pairing process between devices.

Advantages:

- Ideal for battery-powered devices due to low energy requirements.

- Simplified protocols for point-to-point communication.
- Cost-effective and widely available.

Disadvantages:

- Limited range compared to Wi-Fi and Zigbee.
- Lower data transfer rates (up to 2 Mbps with BLE).

Potential Applications in the Project:

- Short-range communication between the robot and other devices.
- Real-time feedback systems in localized environments (not used in this project due to range constraints).

Zigbee

Overview: Zigbee is a low-power, low-data-rate wireless communication protocol optimized for IoT applications, particularly in mesh networking.

Key Features:

- Mesh network support, allows devices to communicate over large areas by relaying messages.
- Very low power consumption.
- Designed for simple IoT networks like smart homes and sensor systems.

Advantages:

- Efficient for low-bandwidth communication in distributed systems.
- Reliable communication in large-scale deployments using mesh topology.
- Secure with AES encryption.

Disadvantages:

- Slower data transfer rates (~250 Kbps).
- Requires specialized hardware and software stack.

Potential Applications in the Project:

- Could be used for distributed sensor systems in larger-scale robotic applications.

The rationale for Choosing Wi-Fi

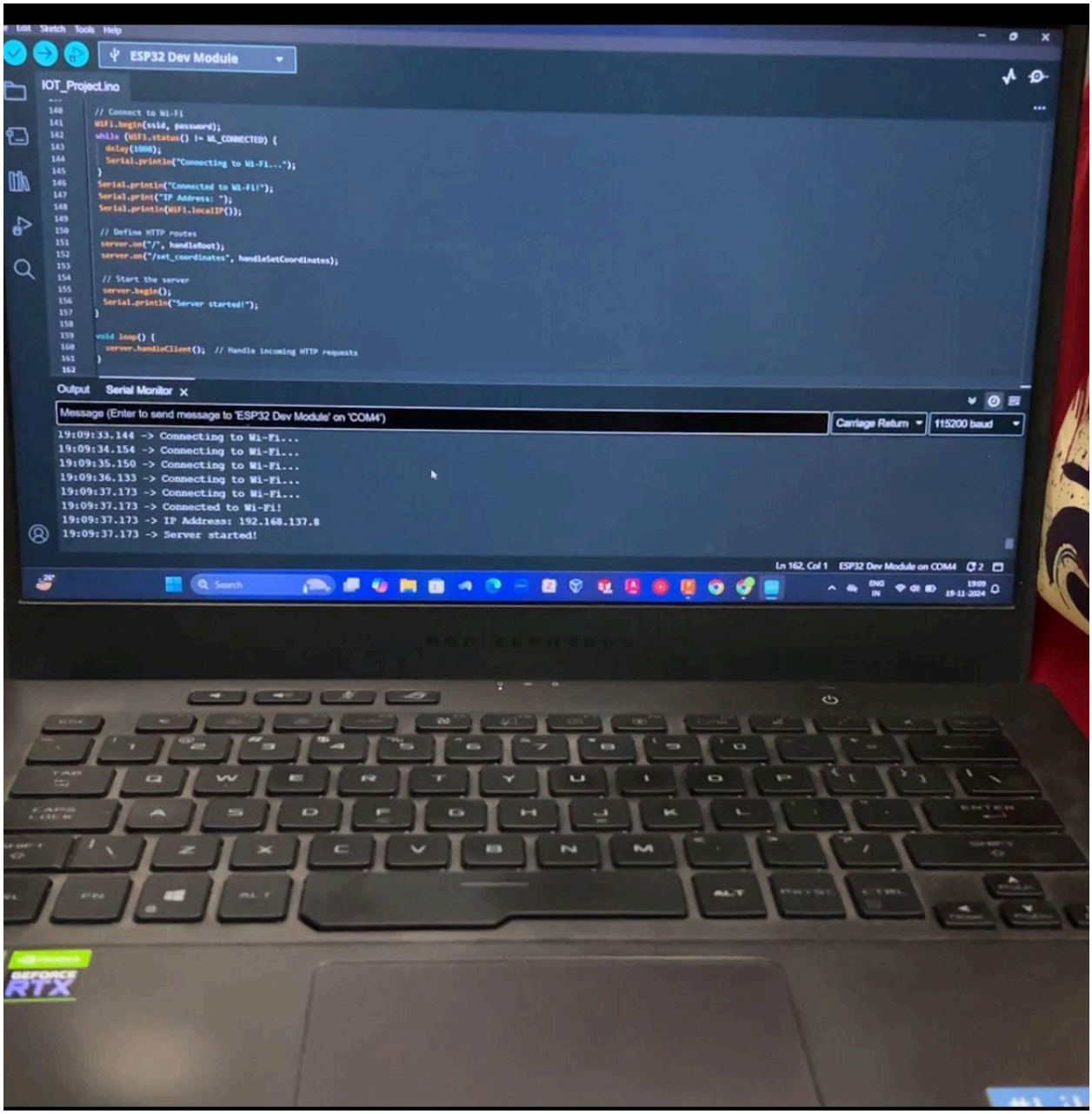
- **Versatility**: Wi-Fi supports both HTTP and WebSocket, offering flexibility for both command execution and real-time monitoring.
- **Range**: Ideal for the indoor environment of the project (~50 meters range).
- **Compatibility**: Easily integrates with ESP32, laptop, and other IoT devices without additional hardware.

Chapter 11: Details of Supporting Tools

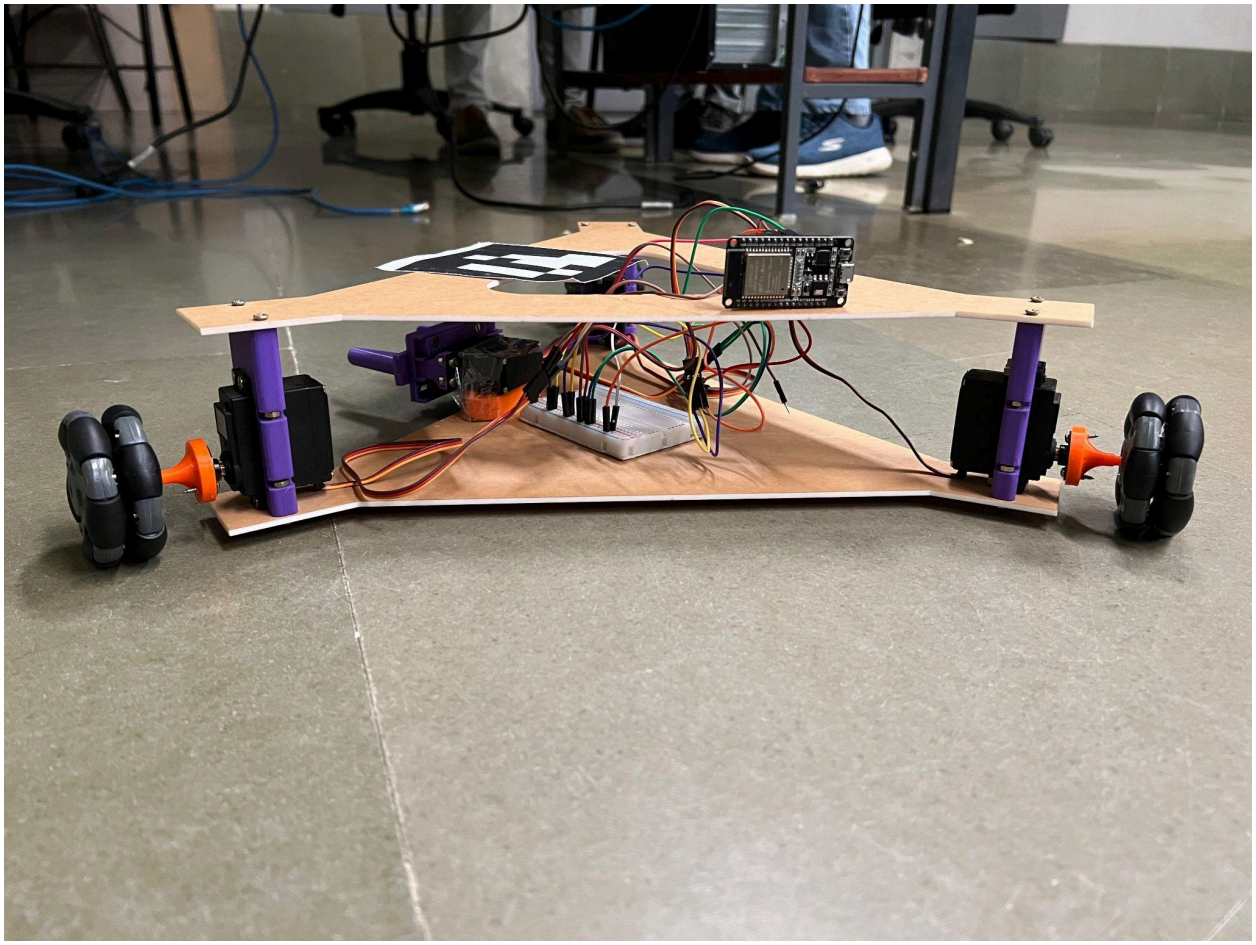
Tool/Framework	Purpose	Key Role in the Project
Python	Core programming language	Main development for marker detection, and control logic.
Flask	Lightweight web framework	Hosting the control interface for servo and robot control.
OpenCV	Image processing and computer vision	Detecting and tracking ArUco markers.
ESP32	Microcontroller with Wi-Fi	Receiving commands and controlling the servo motor.
Requests	HTTP requests library	Communication between laptop and ESP32.
Tkinter	GUI library	Start button for initiating the process.
Arduino IDE	Microcontroller programming environment	ESP32 configuration and logic development.

Chapter 12: Complete Program

Internet Connected



Simulation Image



Robot & Object Tracking with Communication to ESP32

```
import cv2 as cv
from cv2 import aruco
import numpy as np
import math
import requests
import tkinter as tk
from threading import Thread


# ESP32 Configuration
ESP32_IP = "192.168.137.8"  # Replace with your ESP32's IP address


# Load in the calibration data
calib_data =
np.load(r"C:\Users\viran\Downloads\OpenCV-main\Wheelin-Dealing-Bots\4.
calib_data\MultiMatrix.npz")


# Extract the camera calibration matrices
cam_mat = calib_data["camMatrix"]
dist_coef = calib_data["distCoef"]


MARKER_SIZE = 6  # Size of the markers in cm


# Define the known coordinates of each corner marker
marker_coords = {
    0: (0, 0),      # Marker 0 at (0, 0)
    1: (150, 0),    # Marker 1 at (150, 0)
    2: (150, -60),  # Marker 2 at (150, -60)
```

```
    3: (0, -60),    # Marker 3 at (0, -60)
}

# Define marker IDs for the robot and the object
ROBOT_MARKER_ID = 4
OBJECT_MARKER_ID = 5

# Define the dictionary for ArUco markers
marker_dict = aruco.getPredefinedDictionary(aruco.DICT_5X5_250)
param_markers = aruco.DetectorParameters()

# Function to map marker positions to real-world coordinates
def get_real_world_coordinates(tVec, H):
    point_camera = np.array([tVec[0][0], tVec[0][1], 1.0])  # (x, y, 1) in the camera frame
    point_world = np.dot(H, point_camera.T)  # Transform to the world frame
    return point_world[0] / point_world[2], point_world[1] / point_world[2]  # Normalize by z

# Function to send coordinates to ESP32
def send_coordinates_to_esp(robot_position, object_position):
    try:
        response = requests.get(
            f"http://{ESP32_IP}/set_coordinates",
            params={
                "robotX": robot_position[0],
                "robotY": robot_position[1],
                "objectX": object_position[0],
```

```
                "objectY": object_position[1],
            },
        )
        if response.status_code == 200:
            print("Coordinates sent successfully!")
        else:
            print(f"Failed to send coordinates: {response.status_code},
{response.text}")
    except Exception as e:
        print(f"Error sending coordinates to ESP32: {e}")


# Function to track and send coordinates
def track_and_send():
    cap = cv.VideoCapture(0)
    robot_position = None
    object_position = None
    homography_matrix = None

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        gray_frame = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
        marker_corners, marker_IDs, _ = aruco.detectMarkers(
            gray_frame, marker_dict, parameters=param_markers
        )
```

```
        if marker_corners:
            rVec, tVec, _ = aruco.estimatePoseSingleMarkers(
                marker_corners, MARKER_SIZE, cam_mat, dist_coef
            )

            camera_points = []
            world_points = []

            for i, (ids, corners) in enumerate(zip(marker_IDs,
marker_corners)):
                marker_id = ids[0]

                if marker_id in marker_coords:
                    camera_points.append([tVec[i][0][0], tVec[i][0][1]])
                    world_points.append(marker_coords[marker_id])

            if len(camera_points) == 4:
                camera_points = np.array(camera_points, dtype=np.float32)
                world_points = np.array(world_points, dtype=np.float32)
                homography_matrix, _ = cv.findHomography(camera_points,
world_points)

            for i, (ids, corners) in enumerate(zip(marker_IDs,
marker_corners)):
                marker_id = ids[0]

                if marker_id == ROBOT_MARKER_ID and homography_matrix is
not None:
                    robot_position = get_real_world_coordinates(tVec[i],
homography_matrix)
```

```
                if marker_id == OBJECT_MARKER_ID and homography_matrix is
not None:
                    object_position = get_real_world_coordinates(tVec[i],
homography_matrix)

        if robot_position and object_position:
            print(f"Robot Position: {robot_position}, Object Position:
{object_position}")
            send_coordinates_to_esp(robot_position, object_position)
            break

        cv.imshow("Robot and Object Tracking", frame)
        if cv.waitKey(1) & 0xFF == ord("q"):
            break

    cap.release()
    cv.destroyAllWindows()

# Function to launch tracking in a separate thread
def start_tracking():
    tracking_thread = Thread(target=track_and_send, daemon=True)
    tracking_thread.start()

# GUI for Start Button
def create_gui():
    root = tk.Tk()
    root.title("Robot and Object Tracker")
```

```
    tk.Label(root, text="Robot and Object Tracking", font=("Arial",
16)).pack(pady=10)


    start_button = tk.Button(
        root,
        text="Start",
        font=("Arial", 14),
        bg="green",
        fg="white",
        command=start_tracking
    )
    start_button.pack(pady=20)


    root.mainloop()


# Run the GUI
if __name__ == "__main__":
    create_gui()
```

Code for ESP to receive Data from Camera & Rotating Servo motors with Omni-directional wheels

```
#include <WiFi.h>
#include <WebServer.h>
#include <ESP32Servo.h>
#include <math.h>

// Wi-Fi credentials
const char* ssid = "Hotspot";
const char* password = "hi123456";

// Create a web server on port 80
WebServer server(80);

// Servo objects for the wheels and gripper
Servo servo1, servo2, servo3, gripperServo;

// GPIO pins for the servos
const int servoPin1 = 17; // Wheel 1
const int servoPin2 = 18; // Wheel 2
const int servoPin3 = 19; // Wheel 3
const int gripperPin = 16; // Gripper servo on GPIO 16

// Constants for the robot
const float wheelRadius = 0.029;    // Radius of the wheel in meters (29
mm)
const float robotRadius = 0.1;      // Distance from the center to the
wheel in meters (100 mm)
const int stopPulse = 1500;         // Neutral position
const int maxPWMRange = 400;        // Max PWM range for full speed
adjustment

// Gripper angles
const int gripperMinAngle = -65; // Minimum gripper angle
```

```
const int gripperMaxAngle = 65;  // Maximum gripper angle

// Robot positions
float currentX = 0.0, currentY = 0.0; // Robot's initial position
float targetX = 0.0, targetY = 0.0;   // Target position

// Function to calculate the PWM signal for each servo
void moveRobot(float Vx, float Vy, float omega) {
  // Calculate wheel velocities
  float v1 = (Vx) / wheelRadius;                          // Wheel 1
velocity
  float v2 = (-sqrt(3) / 2 * Vx - 0.5 * Vy) / wheelRadius; // Wheel 2
velocity
  float v3 = (sqrt(3) / 2 * Vx - 0.5 * Vy) / wheelRadius;  // Wheel 3
velocity

  // Convert wheel velocities to PWM signals
  int pwm1 = stopPulse + constrain(v1 * maxPWMRange, -maxPWMRange,
maxPWMRange);
  int pwm2 = stopPulse + constrain(v2 * maxPWMRange, -maxPWMRange,
maxPWMRange);
  int pwm3 = stopPulse + constrain(v3 * maxPWMRange, -maxPWMRange,
maxPWMRange);

  // Write PWM signals to the servos
  servo1.writeMicroseconds(pwm1);
  servo2.writeMicroseconds(pwm2);
  servo3.writeMicroseconds(pwm3);
}

// Function to move the gripper for 4 seconds
void moveGripper() {
  Serial.println("Moving the gripper...");
  unsigned long startTime = millis();
  while (millis() - startTime < 4000) { // Run for 4 seconds
```

```
    for (int angle = gripperMinAngle; angle <= gripperMaxAngle; angle++) {
      gripperServo.write(90 + angle); // Offset by 90° for standard servo
range
      delay(20);
      if (millis() - startTime >= 4000) break; // Stop if 4 seconds
elapsed
    }
    for (int angle = gripperMaxAngle; angle >= gripperMinAngle; angle--) {
      gripperServo.write(90 + angle);
      delay(20);
      if (millis() - startTime >= 4000) break; // Stop if 4 seconds
elapsed
    }
  }
  Serial.println("Gripper movement completed.");
}

// Function to move the robot to target coordinates for 4 seconds
void moveToCoordinates() {
  Serial.println("Starting robot movement...");
  unsigned long startTime = millis();

  // Calculate velocities to reach the target
  float deltaX = targetX - currentX;
  float deltaY = targetY - currentY;

  // Normalize direction
  float distance = sqrt(deltaX * deltaX + deltaY * deltaY);
  float Vx = (deltaX / distance) * 0.2; // Constant speed of 0.2 m/s
  float Vy = (deltaY / distance) * 0.2;

  // Move the robot for 4 seconds
  while (millis() - startTime < 4000) {
    moveRobot(Vx, Vy, 0); // Move with calculated velocity
    delay(100);
```

```
    // Update the current position (simulate encoder feedback)
    currentX += Vx * 0.1; // Assumes 0.1 seconds per loop
    currentY += Vy * 0.1;

    // Debugging: Print current position
    Serial.printf("Current Position: X = %.2f, Y = %.2f\n", currentX,
currentY);
  }

  // Stop the robot
  moveRobot(0, 0, 0);
  Serial.println("Robot movement stopped after 4 seconds!");

  // Move the gripper
  moveGripper();
}

// HTTP handler for the root endpoint
void handleRoot() {
  server.send(200, "text/plain", "ESP32 is ready to receive commands!");
}

// HTTP handler for setting initial and target coordinates
void handleSetCoordinates() {
  if (server.hasArg("robotX") && server.hasArg("robotY") &&
server.hasArg("objectX") && server.hasArg("objectY")) {
    // Parse coordinates from the HTTP request
    currentX = server.arg("robotX").toFloat();
    currentY = server.arg("robotY").toFloat();
    targetX = server.arg("objectX").toFloat();
    targetY = server.arg("objectY").toFloat();

    // Debugging: Print received coordinates
```

```
    Serial.printf("Initial Position: X = %.2f, Y = %.2f\n", currentX,
currentY);
    Serial.printf("Target Position: X = %.2f, Y = %.2f\n", targetX,
targetY);

    // Acknowledge receipt
    server.send(200, "text/plain", "Coordinates received!");

    // Move the robot
    moveToCoordinates();
  } else {
    server.send(400, "text/plain", "Missing required parameters!");
  }
}

void setup() {
  Serial.begin(115200);

  // Attach servos
  servo1.attach(servoPin1, 500, 2500);
  servo2.attach(servoPin2, 500, 2500);
  servo3.attach(servoPin3, 500, 2500);
  gripperServo.attach(gripperPin, 500, 2500);

  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to Wi-Fi...");
  }
  Serial.println("Connected to Wi-Fi!");
  Serial.print("IP Address: ");
  Serial.println(WiFi.localIP());

  // Define HTTP routes
```

```
  server.on("/", handleRoot);
  server.on("/set_coordinates", handleSetCoordinates);

  // Start the server
  server.begin();
  Serial.println("Server started!");
}

void loop() {
  server.handleClient();  // Handle incoming HTTP requests
}
```

Appendix A: Datasheets

1. Logitech C930e Webcam (For Aruco Marker Detection)

Key Specifications (from Datasheet):

- **Resolution**: Full HD 1080p (1920 x 1080)
- **Field of View**: 90° (Diagonal)
- **Frame Rate**: 30fps
- **Zoom**: 4x digital zoom in Full HD
- **Autofocus**: Yes
- **Connection**: USB 2.0
- **Dimensions**: 43.3mm x 94mm x 71mm
- **Operating Voltage**: 5V (via USB)
- **Compatibility**: Plug-and-play with Raspberry Pi (via USB)

DataSheet

2. IR Sensor (TCRT5000) Mounted on the Gripper

Key Specifications (from Datasheet):

- **Operating Voltage**: 3.3V to 5V
- **Detection Range**: 1mm to 25mm (depending on the surface reflectivity)
- **Current Consumption**: 10mA to 20mA
- **Response Time**: < 1ms
- **Sensing Method**: Infrared reflection
- **Output**: Digital (HIGH when no object detected, LOW when object detected)
- **Dimensions**: 10mm x 30mm x 15mm

DataSheet

3. Servo Motor (TowerPro MG995)

Key Specifications (from Datasheet):

- **Operating Voltage**: 4.8V to 6.6V
- **Torque**:
  - 10kg/cm at 4.8V
  - 12kg/cm at 6V
- **Speed**:
  - 0.2 sec/60° at 4.8V
  - 0.16 sec/60° at 6V
- **Rotation Range**: 180°
- **Dimensions**: 40.7mm x 19.7mm x 42.9mm
- **Weight**: 55g
- **PWM Signal**: 1ms to 2ms for 0° to 180° rotation
- **Operating Temperature**: -30°C to +60°C

DataSheet

4. Raspberry Pi 4 (4GB RAM) - Central Controller

Key Specifications (from Datasheet):

- **Processor**: Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- **Memory**: 4GB LPDDR4-3200 SDRAM
- **Connectivity**:
  - Wi-Fi 802.11ac, Bluetooth 5.0, Gigabit Ethernet
  - 2 x USB 3.0, 2 x USB 2.0 ports
- **Camera Support**: Via CSI (Camera Serial Interface) or USB
- **GPIO Pins**: 40-pin GPIO header
- **Power Supply**: 5V DC via USB-C or GPIO pins

DataSheet

5. Flask Web Server (Software)

Although this is software and not a hardware component, here's a brief overview of its capabilities:

- **Type**: Python web server framework
- **Operating Environment**: Runs on the Raspberry Pi (Linux OS)
- **Connectivity**: Provides a web interface for controlling the robot and displaying real-time sensor data
- **Integration**: Easily integrates with Python scripts controlling the GPIO for hardware components (e.g., servo motor and sensors)

Documentation

Appendix B: Programming Review

Feature	C	Python	Java	MATLAB
Type	Compiled	Interpreted	Compiled	Interpreted
Speed	Very high (close to hardware)	Moderate (due to interpretation)	High (JVM optimized)	Low (due to interpretation overhead)
Ease of Use	Complex (low-level programming)	Easy (high-level, dynamic typing)	Moderate (object-oriented)	Easy (specialized for mathematical computations)
Memory Management	Manual (e.g., malloc, free)	Automatic (garbage collection)	Automatic (garbage collection)	Automatic
Best Applications	Embedded systems, real-time systems	Rapid prototyping, web development, AI	Enterprise software, mobile apps	Mathematical modeling, simulation
Learning Curve	Steep	Low	Moderate	Low (if math background is strong)
Hardware Interaction	Direct, efficient	Indirect (via libraries like PySerial)	Indirect	Indirect
Library Support	Limited	Extensive (e.g., NumPy, OpenCV)	Extensive (e.g., Spring, Hibernate)	Limited to domain-specific toolboxes
Portability	High (cross-platform with compilers)	High (cross-platform with interpreter)	High	Low (MATLAB license dependency)
Cost	Free (open-source compilers)	Free (open-source)	Free (open-source alternatives)	Expensive (proprietary software)

Appendix C: Trouble-Shooting

Example: Debugging Marker Detection and Alignment

Problem Encountered:
During the initial implementation, the camera failed to detect all four corner markers simultaneously. This caused inconsistencies in the homography matrix, leading to incorrect real-world coordinate mapping. Additionally, the detected markers were misaligned, affecting the robot's position tracking.

Steps Taken to Solve the Problem:

1. **Camera Placement and Field of View**:
    - Adjusted the camera height and angle to ensure all four corner markers were within the frame at all times.
    - Increased the distance between markers to avoid overlap or partial visibility.
2. **Calibration and Thresholding**:
    - Performed camera calibration to correct lens distortion using OpenCV's calibration module.
    - Fine-tuned image preprocessing (grayscale conversion and adaptive thresholding) to improve marker detection in varying lighting conditions.
3. **Testing and Debugging**:
    - Used debug visuals to overlay detected marker IDs and corner coordinates on the camera feed.
    - Logged marker coordinates and transformation errors to identify and fix discrepancies.
4. **Optimization**:
    - Adjusted the detection parameters (param_markers) to improve marker detection speed and accuracy.
    - Used a test environment with consistent lighting to stabilize detection during testing.

Outcome:
The adjustments resolved the detection issues, enabling consistent tracking of all markers. The robot's path tracing and real-world coordinate mapping became accurate and reliable.

Appendix D: Real-Life Mounting of Your System

Deployment Location:
The system is designed for use in warehouses, manufacturing plants, or indoor logistical facilities where robots handle material movement and tracking tasks.

Mounting and Protection:

1. **Camera Mounting**:
    - Positioned overhead on a secure, vibration-free mount (e.g., ceiling brackets or industrial tripods).
    - Encased in a weatherproof and dustproof enclosure to protect against environmental factors such as dust, moisture, or temperature fluctuations.
2. **Microcontroller and Servo Motor**:
    - Placed in protective industrial-grade enclosures made from shockproof and waterproof materials.
    - Located in areas with minimal exposure to mechanical or environmental risks.
3. **Infrastructure Protection**:
    - Use physical barriers or fences to prevent accidental damage from workers or machinery.
    - Implement surveillance cameras and restricted access to prevent vandalism or intentional harm.

Power Supply and Connectivity:

1. **Power Supply**:
    - Use uninterruptible power supplies (UPS) to ensure consistent operation during power outages.
    - Integrate backup power systems such as solar panels or batteries for prolonged downtime.
2. **Internet Connectivity**:
    - Deploy high-speed Wi-Fi routers for stable communication.
    - Use cellular connectivity (4G/5G) as a fallback to ensure uninterrupted control in case of local network issues.

Appendix E: Real Life Scenarios

Challenge 1: Network Latency

Problem:
In real-world environments, network congestion or interference may cause delays in communication between the laptop and ESP32, affecting real-time control and feedback.

Solution:

- Use dual-band Wi-Fi routers to reduce interference and prioritize robot communication.
- Introduce message acknowledgment protocols to confirm data delivery and prevent missed commands.

Challenge 2: Hardware Durability

Problem:
In a warehouse or factory, the robot and components may be exposed to dust, heat, or accidental collisions, affecting their durability.

Solution:

- Use industrial-grade hardware that can withstand harsh environments.
- Regularly inspect and maintain components to ensure long-term functionality.
- Apply protective coatings or housings to shield sensitive parts from environmental wear and tear.

Challenge 3: Scaling the System

Problem:
As more robots are introduced into the workspace, managing simultaneous communication and avoiding collisions becomes complex.

Solution:

- Use a centralized coordination system to assign unique tasks and manage robot paths dynamically.
- Integrate collision avoidance algorithms to ensure safe navigation in crowded environments.
- Employ mesh networking protocols like Zigbee to handle communication for multiple robots efficiently.

References

S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed., Upper Saddle River, NJ, USA: Prentice Hall, 2010.

- For theoretical concepts related to path planning and robot localization.

G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, vol. 25, no. 11, pp. 120-125, Nov. 2000.

- Used for marker detection and pose estimation.

IEEE Computer Society, "IEEE Standard for Wi-Fi Wireless Communications," *IEEE Std 802.11-2020*, Jan. 2020.

- Specifications and details of Wi-Fi communication used in the project.

Arduino, "ESP32: Wi-Fi and Bluetooth Development Board," *Arduino Documentation*, [Online]. Available: https://www.arduino.cc/en/Guide/ESP32. [Accessed: 19-Nov-2024].

- Reference for programming and configuring ESP32 microcontrollers.

Flask Framework, "Flask Documentation," [Online]. Available: https://flask.palletsprojects.com. [Accessed: 19-Nov-2024].

- Reference for building the web interface to control the servo motor and monitor the system.