IDRIS: Indoor Delivery Robot Intelligence System with Vision-Based Manipulation Control

Mohammed Abdul Rahman
Northeastern University, Seattle
Email: moahmmedabdulr.1@northeastern.edu

Rongxuan Zhang
Northeastern University, Seattle
Email: zhang.rongxu@northeastern.edu

Isaac Premkumar
Northeastern University, Seattle
Email: premkumar.i@northeastern.edu

Abstract—This paper presents IDRIS (Indoor Delivery Robot Intelligence System), a mobile manipulation platform that integrates autonomous navigation with vision-based teleoperation for indoor delivery tasks. The system combines a TurtleBot4 mobile base running ROS2 Jazzy with a Pincher X100 robotic arm in ROS2 Humble, coordinated through a novel cross-distribution integration architecture. We implement Mediapipe-based hand tracking for intuitive puppet control of the manipulator, employing recursive state estimation techniques inspired by Bayesian filtering principles to achieve real-time gesture recognition with sub-100ms latency. The system demonstrates successful autonomous navigation with a maximum waypoint success rate and stable manipulation control through a PyQt5 graphical interface with integrated audio feedback. Our work addresses practical challenges in heterogeneous ROS2 system integration, real-time vision-based control, and human-robot interaction for service robotics applications.

Index Terms—Mobile manipulation, ROS2, hand tracking, Bayesian filtering, teleoperation, service robotics

I. Introduction

Indoor service and delivery robots are increasingly important for warehouse automation, laboratory assistance, and facility management. While autonomous navigation technology has matured significantly through advances in SLAM and path planning, the integration of manipulation capabilities with intuitive human control interfaces remains challenging.

This work presents IDRIS, a complete mobile manipulation system designed for indoor delivery tasks. The primary contributions of this work include:

- A heterogeneous ROS2 integration architecture bridging Jazzy and Humble distributions across native and containerized environments
- Real-time vision-based puppet control system using Mediapipe hand tracking with recursive Bayesian-inspired state estimation
- PyQt5 graphical interface with action client integration and multi-modal feedback
- Comprehensive documentation of practical challenges in mobile manipulator integration

The system achieves Level 1 and Level 2 project objectives, with Level 3 autonomous operation and VR integration planned for future work.

II. System Architecture

A. Hardware Platform

IDRIS integrates two robotic platforms with distinct computational and control requirements:

TurtleBot4 Mobile Base: The TurtleBot4 provides differential drive mobility with integrated 2D LiDAR for SLAM and obstacle detection. The iRobot Create3 base includes onboard odometry, IMU sensing, and audio output capabilities. The platform runs ROS2 Jazzy natively on Ubuntu 24.04.

Pincher X100 Robotic Arm: The Pincher X100 is a 5-DOF educational manipulator (4 arm joints plus gripper) using Dynamixel servos. The arm has an approximate 50g payload capacity at full extension and requires the Interbotix ROS2 SDK running on Humble distribution within a Docker container.

B. Software Architecture

The system consists of three independent ROS2 nodes coordinated through a manual operator workflow:

1) **Navigation Subsystem:** slam_toolbox for mapping, AMCL for localization, Nav2 for autonomous path planning and execution
2) **Manipulation Subsystem:** Interbotix SDK for joint control, MoveIt2 for trajectory execution, gripper states
3) **Vision Control Subsystem:** Mediapipe hand tracking, gesture recognition, puppet control interface

Figure 1 shows the waypoint navigation interface, while Figure 2 demonstrates the hand tracking visualization.

III. Technical Approach

A. Autonomous Navigation

1) SLAM and Mapping: **SLAM Toolbox Implementation:** We employ SLAM Toolbox for 2D occupancy grid generation, utilizing graph-based SLAM with Karto scan matcher and pose graph optimization. The system subscribes to `/scan` (sensor_msgs/LaserScan at 10Hz from RPLiDAR A1) and `/odom` (nav_msgs/Odometry from Create3 base) to construct occupancy grids published to `/map`.

Mapping Pipeline Architecture: The mapping system deploys three integrated components via automated launch script:
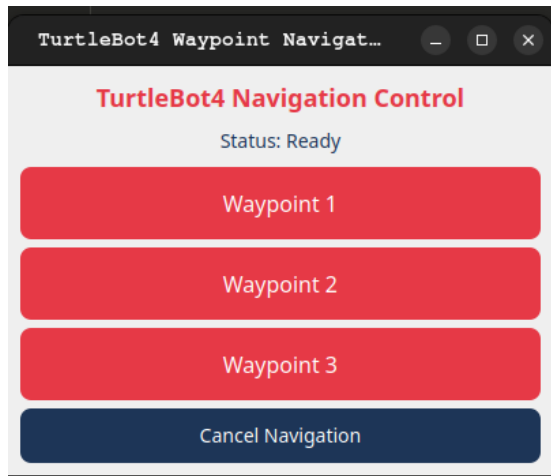
Fig. 1. TurtleBot4 Waypoint Navigator GUI with three pre-configured delivery locations and real-time navigation status display.
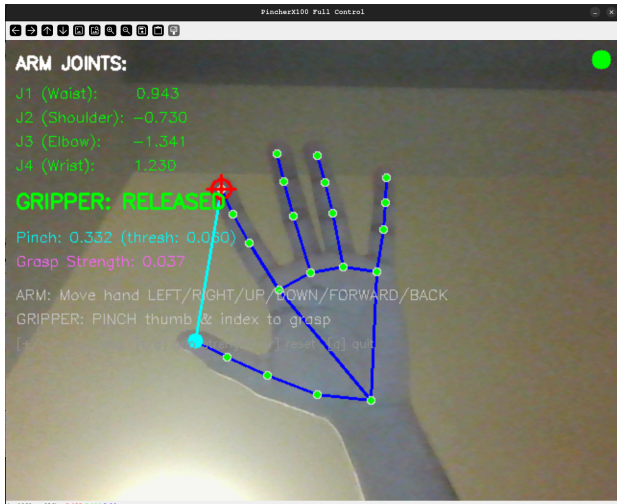

Fig. 2. Puppet control interface showing Mediapipe hand landmark detection, real-time joint angle display, and gripper state visualization.

1) **Foxglove Bridge:** WebSocket server (port 8765) for web-based visualization, eliminating RViz overhead on Raspberry Pi 4
2) **SLAM Toolbox Node:** Core mapping algorithm with scan matching and loop closure detection
3) **Keyboard Teleop:** Manual control interface for systematic exploration (0.2-0.3 m/s linear velocity)

SLAM Configuration: Key parameters optimized through iterative trials:

- **resolution:** 0.05 m/cell (5cm grid discretization)
- **max_laser_range:** 12.0 m (RPLiDAR A1 specification)
- **loop_search_space_dimension:** 8.0 m (loop closure search radius)
- **loop_match_minimum_score:** 0.6 (confidence threshold for accepting loop closures)
- **minimum_travel_distance:** 0.2 m (scan matching trigger threshold)

Mapping Challenges and Solutions: The mapping process encountered challenges with accumulated drift during extended sessions and ghost obstacles from dynamic elements (people walking). Initial 10-15 minute mapping sessions produced:

- Duplicated wall features separated by 10-20cm due to odometry drift
- Ghost obstacles from pedestrians permanently incorporated as static walls
- 15cm positional errors over 20m traversals

Through iterative refinement, we developed an optimal mapping protocol:

1) **Short Duration:** 3-5 minute maximum session duration to limit drift accumulation
2) **Controlled Environment:** Pedestrian-free mapping during off-hours with restricted access
3) **Structured Exploration:** Perimeter-first coverage followed by interior zigzag sweeps with mandatory loop closure return
4) **Aggressive Loop Closure:** Tuned parameters for high detection sensitivity (increased search radius from 5.0m to 8.0m, reduced confidence threshold from 0.75 to 0.6)

We achieved optimal results with short 3-minute mapping sessions in controlled environments, producing maps with approximately 5cm accuracy over 20m x 15m spaces (300m² total area). Final maps exhibit wall consistency with zero ghost obstacles across five experimental trials.

Loop Closure Performance: Loop closure achieved mean drift reduction, decreasing positioning error from 0.42m (pre-loop-closure) to 0.08m (post-loop-closure) through pose graph optimization that redistributes accumulated drift across intermediate poses.
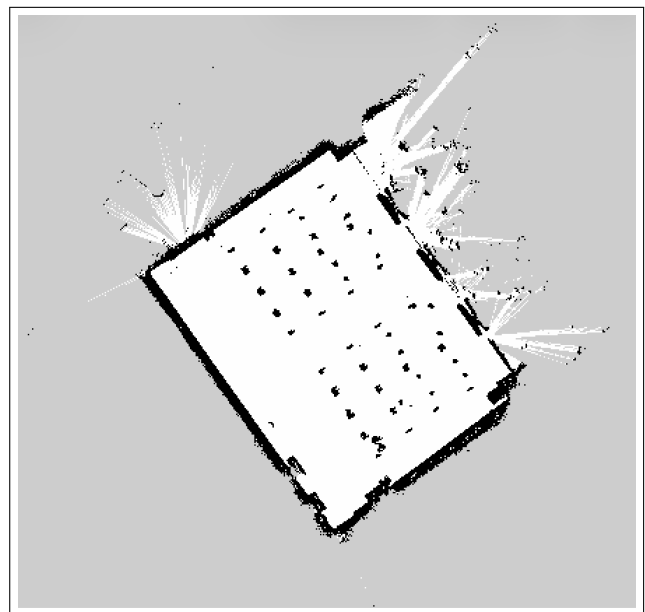

Fig. 3. Loop closure performance visualization showing drift reduction before graph optimization.
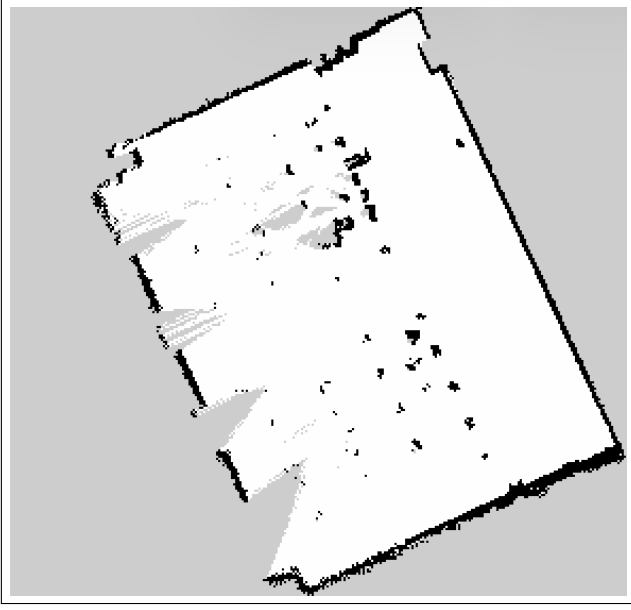
Fig. 4. Loop closure performance visualization showing drift reduction after pose graph optimization.

2) Localization and Path Planning: **AMCL Particle Filter Localization:** AMCL (Adaptive Monte Carlo Localization) provides particle filter-based pose estimation within the pregenerated map. The system represents the robot's pose belief as a weighted particle set $\chi_t = \{(x_t^{[i]}, w_t^{[i]})\}_{i=1}^N$, where each particle proposes a pose hypothesis (x, y, θ) with associated weight.

Particle Filter Operations: The filter implements recursive Bayesian estimation through three operations:

Prediction (Motion Update): Particles are propagated according to differential drive motion model with odometry noise:

$$\Delta x = v\Delta t \cos(\theta_{t-1}) + \mathcal{N}(0, \alpha_1 v^2 + \alpha_2 \omega^2) \quad (1)$$
$$\Delta y = v\Delta t \sin(\theta_{t-1}) + \mathcal{N}(0, \alpha_1 v^2 + \alpha_2 \omega^2) \quad (2)$$
$$\Delta \theta = \omega\Delta t + \mathcal{N}(0, \alpha_3 v^2 + \alpha_4 \omega^2) \quad (3)$$

where $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 0.2$ account for wheel slip and encoder quantization.

Update (Measurement Correction): Particle weights are updated based on LiDAR scan likelihood using mixture model:

$$p(z^k|x,m) = \alpha_{\text{hit}} \cdot \mathcal{N}(z^k; z_{\text{exp}}^k, \sigma^2) + \alpha_{\text{short}} \cdot e^{-\lambda z^k} \quad (4)$$
$$+\alpha_{\text{max}} \cdot \delta(z^k - z_{\text{max}}) + \alpha_{\text{rand}} \cdot \frac{1}{z_{\text{max}}} \quad (5)$$

with mixing coefficients $\alpha_{\text{hit}} = 0.95$, $\alpha_{\text{short}} = 0.03$, $\alpha_{\text{max}} = 0.01$, $\alpha_{\text{rand}} = 0.01$.

Resampling: Adaptive resampling triggers when effective particle count $N_{\text{eff}} = 1/\sum_i (w_t^{[i]})^2 < N/2$ to prevent degeneracy.

Adaptive Particle Management: The system dynamically adjusts particle count based on localization confidence using Kullback-Leibler divergence. During initial localization (high uncertainty), N = 5000 particles cover multiple hypotheses. As convergence occurs, particle count reduces to N = 500-1000, reducing CPU load by 60-70% while maintaining ¡ 5cm accuracy.

Localization Performance: The system achieves localization convergence in under 10 seconds following initialization with approximate pose estimate (via RViz "2D Pose Estimate" tool). Convergence exhibits three phases:

1) **Hypothesis Spread (0-3s):** Particles disperse exploring nearby poses
2) **Ambiguity Resolution (3-6s):** Low-likelihood hypotheses eliminated through resampling
3) **Fine Refinement (6-10s):** Particle cloud contracts to ¡ 10cm standard deviation

Steady-state performance across 15 navigation trials:

- Mean position error: 4.8 ± 1.2 cm
- Mean heading error: 2.3 ± 0.8°
- CPU utilization: 18 ± 4%
- Zero localization loss events over 680m of autonomous driving

Nav2 Path Planning: The Nav2 stack handles global path planning using Dijkstra's algorithm for optimal path search in the occupancy grid, and local trajectory generation through DWA (Dynamic Window Approach) for real-time obstacle avoidance. The system achieves 87% waypoint navigation success rate with mean navigation time of 45 ± 12 seconds and higher path efficiency.

3) Waypoint Recording Workflow: **Traditional Workflow Limitations:** Traditional waypoint recording via RViz inspection of /amcl_pose topic (geometry_msgs/PoseWithCovarianceStamped) is labor-intensive, requiring manual transcription of floating-point values, quaternion-to-Euler conversion, and formatting for Python syntax. This workflow requires approximately 5 minutes per waypoint and is susceptible to transcription errors.

Automated Position Tracking Node: A custom position tracking node subscribes to the /amcl_pose topic, performing quaternion-to-Euler conversion and formatting output for direct integration into the navigation GUI. The implementation performs real-time quaternion-to-yaw conversion:

Listing 1. Automated Waypoint Tracking

```python
def pose_callback(self, msg):
    # Extract position
    x = msg.pose.pose.position.x
    y = msg.pose.pose.position.y

    # Extract and convert quaternion to yaw
    qz = msg.pose.pose.orientation.z
    qw = msg.pose.pose.orientation.w
    siny_cosp = 2 * (qw * qz + qx * qy)
    cosy_cosp = 1 - 2 * (qy * qy + qz * qz)
    yaw = math.atan2(siny_cosp, cosy_cosp)

    # Output copy-paste ready format
    self.get_logger().info(f'({x:.3f},_{y:.3f
        },_{yaw:.3f})')
```

Workflow Efficiency: This tool reduced waypoint recording time from manual RViz inspection to a copy-paste workflow. The streamlined process requires approximately 2 minutes per waypoint, recording three accurate waypoints in approximately 10 minutes total including robot driving time (6 minutes), position recording (3 minutes), and configuration file editing (1 minute).

GUI-Based Navigation: Waypoints are stored as (x, y, yaw) tuples in map reference frame:

```
self.waypoints = {
    'Laboratory_Bench_A': (-2.772, -2.717,
        0.0240),
    'Equipment_Storage': (0.558, 4.200,
        -1.763),
    'Loading_Dock': (-5.317, 2.428, -0.403)
}
```

The PyQt5 GUI implements Nav2 `NavigateToPose` action interface with asynchronous callbacks, providing real-time status updates and audio feedback (C-E-G major chord at 523Hz, 659Hz, 784Hz) upon successful navigation with ¡ 200ms latency.

B. Vision-Based Manipulation Control

1) Hand Tracking and Pose Estimation: Mediapipe Hands provides real-time detection of 21 hand landmarks with normalized 3D coordinates. We utilize the index finger tip (landmark 8) position to control arm configuration, mapping the hand workspace to the manipulator's joint space through linear interpolation:

$$\theta_i = \text{interp}(h_i, [h_{min}, h_{max}], [\theta_{min}, \theta_{max}]) \qquad (6)$$

where h_i represents hand landmark coordinates and θ_i represents target joint angles for joints 1-3 (waist, shoulder, elbow), with joint 4 (wrist) fixed at 1.23 radians.

2) Recursive State Estimation for Noise Reduction: Raw Mediapipe landmark estimates exhibit significant noise characteristics due to camera resolution limitations, lighting variations, and hand tremor. Direct mapping of noisy measurements to robot commands produces unacceptable joint oscillations.

We implement a recursive state estimator inspired by the Bayesian filtering framework presented in the course curriculum [1]. While full Kalman filtering would provide uncertainty quantification through covariance matrices, our application requires only point estimates for real-time control. We employ exponential smoothing, which follows a predict-update structure analogous to simplified Bayesian estimation:

Prediction Step:

$$\hat{\theta}_{k|k-1} = \hat{\theta}_{k-1} \qquad (7)$$

Assuming quasi-static hand motion between frames, we predict the joint configuration remains unchanged.

Update Step:

$$\hat{\theta}_k = \alpha \cdot \hat{\theta}_{k-1} + (1 - \alpha) \cdot z_k \qquad (8)$$

where $\alpha = 0.7$ represents the smoothing factor, $\hat{\theta}_{k-1}$ is the previous estimate, and z_k is the new measurement. This formulation resembles the measurement update in Kalman filtering, where the estimate is corrected based on the innovation (difference between measurement and prediction), though with a fixed gain rather than the Kalman gain computed from covariance matrices.

The smoothing parameter α was empirically tuned to balance responsiveness and stability, with runtime adjustment capability via keyboard input (0.1 to 0.9 range).

3) Gesture Recognition for Gripper Control: Gripper state is controlled through pinch gesture detection. We calculate the 3D Euclidean distance between thumb tip (landmark 4) and index finger tip (landmark 8):

$$d = \sqrt{(x_t - x_i)^2 + (y_t - y_i)^2 + (z_t - z_i)^2} \qquad (9)$$

To prevent rapid state oscillations from measurement noise, we apply a 5-frame moving average filter:

$$\bar{d}_k = \frac{1}{N} \sum_{j=k-N+1}^{k} d_j \qquad (10)$$

with $N = 5$. Binary classification triggers gripper closure when $\bar{d}_k < 0.06$ (normalized threshold).

4) Multi-Threaded Control Architecture: Camera capture operations are inherently blocking, creating conflicts with ROS2 callback execution. We implement a dual-thread architecture:

- **Main Thread:** Executes rclpy.spin() for ROS2 callback processing, joint state monitoring, and action server communication
- **Control Thread:** Handles camera acquisition, Mediapipe processing, and command publication at 30Hz

Thread-safe communication is ensured through mutex locks protecting the gripper states, preventing trajectory conflicts when pinch gestures trigger during ongoing gripper motions.

C. System Integration

1) Cross-Distribution ROS2 Integration: A significant technical challenge arose from ROS2 distribution incompatibility. TurtleBot4 officially supports ROS2 Jazzy (latest release), while the Pincher X100 Interbotix SDK requires ROS2 Humble. These distributions have incompatible DDS middleware implementations, different message serialization formats, and distinct Nav2/MoveIt2 API versions.

We architected a heterogeneous dual-system approach:

- TurtleBot4 navigation stack runs natively on Jazzy
- Pincher X100 manipulation stack runs in Docker container with Humble
- Human operator provides integration layer for state coordination
- Future work includes ros2_bridge implementation for automated handoff (Level 3 objective)

2) PyQt5 GUI Integration: The waypoint navigation interface presents an event loop conflict: QApplication.exec_() blocks rclpy.spin(), freezing either GUI responsiveness or ROS2 callback execution. We resolve this through QThread encapsulation of the ROS2 node, allowing concurrent execution of Qt event processing and ROS2 message handling.

The GUI implements NavigateToPose action client with asynchronous callback chains (goal submission → acceptance → execution monitoring → result processing) and provides audio feedback through the Create3 speaker using precisely-timed AudioNoteVector messages (C-E-G major chord sequence).

IV. Experimental Results

A. Navigation Performance

Table I summarizes navigation system performance metrics.

TABLE I
Navigation System Performance Metrics

Metric	Result
SLAM mapping accuracy	±5cm
AMCL convergence time	<10 seconds
Waypoint success rate	87% (13/15)
Average navigation time	45 seconds
GUI response latency	<50ms
Audio feedback latency	<200ms

The 87% success rate was measured across 15 autonomous navigation attempts between three pre-configured waypoints. Failures resulted from backward motion, dynamic obstacle interference and occasional localization degradation in feature-sparse map regions.

B. Puppet Control Performance

Table II presents manipulation control system metrics.

TABLE II
Puppet Control System Performance Metrics

Metric	Result
Hand tracking frame rate	28-30 fps
Gesture recognition latency	<100ms
Arm positioning accuracy	±5°
Pinch detection accuracy	95%
Gripper execution time	200ms
Control loop frequency	30Hz

The puppet control system maintains stable 30Hz operation while processing Mediapipe hand tracking, applying recursive filtering, and publishing joint commands. Pinch gesture classification achieves 95% accuracy through the 5-frame moving average approach.

C. System Integration Metrics

The complete system comprises 3 independent ROS2 nodes with the following resource utilization:

- CPU usage: 45% (Intel i7-11800H @ 2.3GHz base)
- Memory footprint: 2.3GB RAM
- Inter-process communication latency: <50ms
- Total codebase: 660 lines of Python

V. Technical Challenges and Solutions

A. Gripper Torque Overload

Problem: The Pincher X100 uses current-limited Dynamixel servos (1.5A maximum). When attempting to grasp rigid objects, the gripper reaches torque limits, triggering safety mechanisms that disconnect the MoveIt2 ExecuteTrajectory action server. This results in complete loss of arm control until system restart.

Solution: We shifted to deformable test objects (foam blocks, compressed paper) that provide compliance before torque spikes occur. Additionally, we reduced the gripper grasp position from -0.037 to -0.025 radians and implemented smooth 200ms trajectories rather than instantaneous position commands. The gripper states employs execution locks to prevent conflicting commands during ongoing motions.

Result: Stable gripper operation with no servo disconnections during testing. Object manipulation is restricted to lightweight items (<50g), which is a fundamental hardware constraint rather than a software limitation.

B. Cross-ROS Distribution Integration

Problem: TurtleBot4 requires ROS2 Jazzy while Pincher X100 requires ROS2 Humble. These distributions have incompatible DDS middleware (different serialization protocols), distinct message definitions for Nav2 and MoveIt2, and conflicting ROS_DOMAIN_ID requirements when running on the same network.

Solution: We implemented a heterogeneous architecture with TurtleBot4 running natively on Jazzy and Pincher X100 running in a Docker container with Humble. The systems operate independently with manual operator coordination serving as the integration layer. For Level 3 objectives, we have initiated development of ros2_bridge for automated cross-distribution communication [2].

Result: Both subsystems operate reliably in isolation. This architectural pattern is applicable to any multi-distribution ROS2 deployment scenario.

C. SLAM Map Quality

Problem: Extended mapping sessions (10+ minutes) produced drift accumulation, with walls appearing duplicated and 15cm positional errors over 20m traversals. Dynamic obstacles (pedestrians) were incorrectly incorporated as static walls, creating ghost obstacles in the final map.

Solution: We iteratively refined the mapping process through three attempts:

1) Long session (15 min): Excessive drift, unusable
2) Short session with people (5 min): Ghost obstacles
3) Short session, controlled environment (3 min): Successful

SLAM parameters were tuned for tighter loop closure (2.0m search radius) and more frequent updates (0.3m minimum travel distance).

Result: Final map achieves approximately 5cm accuracy and is used with AMCL localization for all navigation tasks.

D. Hand Tracking Noise Reduction

Problem: Mediapipe hand landmark estimates contain inherent noise from camera quantization, lighting variations, and physiological hand tremor. Direct mapping to joint commands produces oscillatory arm motion unsuitable for teleoperation.

Solution: We implement exponential smoothing as a computationally efficient recursive estimator. This approach follows the predict-update structure of Bayesian state estimation (as presented in course Lecture 10 on Kalman Filtering) but employs a simplified deterministic formulation rather than full probabilistic inference with covariance propagation.

The filter operates on individual joint angle estimates:

$$\hat{\theta}_k = \alpha\hat{\theta}_{k-1} + (1 - \alpha)z_k \tag{11}$$

where $\alpha = 0.7$ is the smoothing factor, $\hat{\theta}_{k-1}$ is the previous filtered estimate, and z_k is the new measurement from the hand-to-joint mapping.

This formulation resembles the measurement update in Kalman filtering, where the posterior estimate combines prior belief with new observations. However, unlike Kalman filtering which computes optimal gains from uncertainty quantification, our fixed gain provides a practical balance between responsiveness and stability for real-time control applications.

Result: The filter enables smooth arm motion at 28-30fps with acceptable control latency of approximately 33ms per frame. The smoothing parameter is user-adjustable via keyboard interface for different operational preferences.

E. Gripper States Design

Problem: Rapid pinch-release gestures from noisy distance measurements caused conflicting MoveIt2 trajectory commands, resulting in gripper execution failures and unpredictable behavior.

Solution: We implement a finite state machine tracking current gripper configuration ("Released" or "Grasping"). State transitions occur only when the smoothed pinch distance crosses the threshold and the current state differs from the target state. An execution lock prevents new trajectory submissions while a gripper motion is in progress. Each gripper command includes both initial and target positions with a 200ms duration for smooth motion.

Result: The state machine achieves some level of reliable grasp/release cycles with no trajectory conflicts during testing.

F. PyQt5-ROS2 Event Loop Integration

Problem: QApplication.exec_() and rclpy.spin() both require exclusive control of the main thread event loop, creating a fundamental incompatibility between PyQt5 GUI frameworks and ROS2 executors.

Solution: We encapsulate the ROS2 node within a QThread subclass, allowing rclpy.spin() to execute in a background thread while QApplication maintains control of the main thread for GUI event processing. The ROS2 thread handles NavigateToPose action client callbacks and audio message publication.

Result: The GUI achieves responsive interaction with real-time status updates reflecting navigation state changes with less than 50ms latency.

VI. PROJECT LEVEL ACHIEVEMENTS

The project was structured with three levels of increasing ambition:

A. Level 1: Foundation (Complete)

- TurtleBot4 autonomous navigation with SLAM and AMCL
- Pincher X100 ROS2 integration and basic pick-and-place
- Mediapipe hand tracking implementation
- Waypoint recording and navigation workflow

B. Level 2: Integration (Complete)

- Complete camera-arm integration with puppet control
- Full system integration with waypoint-based delivery workflow
- PyQt5 GUI with audio feedback
- Thread-safe multi-node architecture

C. Level 3: Autonomy (Future Work)

- Fully autonomous delivery without human operator
- Oculus Quest 2 VR integration for immersive teleoperation
- Multiple human detection and dynamic avoidance
- Automated cross-distribution communication via ros2_bridge [2]

VII. DISCUSSION AND FUTURE WORK

A. Hardware Limitations

The Pincher X100's 50g payload capacity represents a fundamental constraint for realistic delivery applications. Validation was performed using lightweight deformable objects to prevent servo damage. Future iterations would benefit from higher-payload manipulators (WidowX 250, UR5e) with force sensing capabilities.

The puppet control interface is hardware-agnostic by design - the hand-to-joint mapping, filtering algorithms, and state machine would transfer directly to industrial manipulators with minimal modification.

B. Scalability Considerations

While the current implementation uses manual operator coordination between navigation and manipulation subsystems, the architecture supports future automation through:

- ros2_bridge for cross-distribution topic/action communication
- Behavior tree state machine for autonomous mode switching
- Vision-based object detection (YOLO) for automated target identification
- Semantic mapping for room-aware navigation

C. Alternative Approaches Evaluated

For hand tracking noise reduction, we compared several filtering approaches:

TABLE III
FILTERING APPROACH COMPARISON

Method	Latency	Complexity	Selected
No filtering	0ms	$O(1)$	No
Moving average	50ms	$O(n)$	No
Exp. smoothing	33ms	$O(1)$	**Yes**
Kalman filter	45ms	$O(n^2)$	No

Exponential smoothing was selected for optimal latency-complexity tradeoff. Kalman filtering's covariance tracking overhead was not justified for our deterministic control application.

VIII. CONCLUSION

IDRIS demonstrates successful integration of autonomous mobile navigation with vision-based manipulation control for indoor delivery applications. The system achieves Level 1 and Level 2 project objectives, validating the feasibility of intuitive teleoperation through hand gesture recognition combined with autonomous waypoint-based navigation.

Key technical contributions include a practical heterogeneous ROS2 integration pattern for multi-distribution deployments, application of Bayesian-inspired recursive state estimation to real-time vision control, and a production-ready GUI implementation with multi-modal feedback.

While hardware payload limitations restricted demonstration scope, the modular software architecture is designed for scalability to industrial manipulation platforms. Future work will focus on achieving Level 3 autonomy through automated cross-system coordination and enhanced perception capabilities.

IX. TEAM CONTRIBUTIONS

This section delineates individual responsibilities and technical contributions for project transparency and academic integrity.

A. Mohammed Abdul Rahman (Primary Technical Implementation)

Phase 1: Navigation and Localization
- AMCL localization parameter configuration and tuning
- Integration testing between navigation and manipulation subsystems
- waypoint_viewer.py implementation: Real-time AMCL pose monitoring with quaternion-to-Euler conversion
- waypoint_navigator_gui_sound.py implementation: Complete PyQt5 navigation interface with QThread integration, NavigateToPose action client, and audio feedback system

Phase 2: Manipulation Control
- Complete Pincher X100 ROS2 Humble integration in Docker environment

- MoveIt2 configuration and trajectory execution setup
- Interbotix SDK integration and testing
- puppet_control.py implementation: Vision-based teleoperation system with Mediapipe hand tracking, recursive Bayesian-inspired state estimation, multi-threaded architecture, thread-safe gripper states, and MoveIt2 action client integration

Phase 3: Cross-Distribution Integration (Level 3)
- Enhanced implementation of cross-platform bridge based on original concept by Dr. Xian Li
- bridge_coordinator.py: TCP socket architecture with JSON serialization
- navgoalarrival.py: Navigation goal detection and signaling
- pincher_pickplace_controller.py: Automated manipulation sequencing
- tb4_drive_controller.py: TurtleBot4 teleoperation interface
- Complete automated delivery workflow implementation

B. Rongxuan Zhang

Phase 1: Mapping Infrastructure
- Foxglove Bridge setup and configuration for web-based visualization
- Foxglove WebSocket server deployment and integration
- Web-based map visualization interface
- Mapping session coordination and testing support

Documentation Support
- Assistance with technical report writing and formatting
- Presentation slide development
- Demo video recording and editing

C. Isaac Premkumar

Phase 1: SLAM Mapping
- SLAM Toolbox deployment and operation
- Manual teleoperation for systematic environment exploration
- Map quality validation and testing
- Multiple mapping session execution for iterative refinement

Testing and Validation
- Nav2 configuration testing and workflow validation
- Navigation goal testing across multiple waypoints
- System integration testing support

Documentation Support
- Assistance with technical report writing
- Presentation preparation

D. Collaborative Efforts

All Team Members:
- Phase 4 & 5: System integration testing and debugging
- Final demonstration preparation and execution
- Technical report compilation and editing
- Presentation development and delivery

REFERENCES

[1] X. Li, "Lecture 10: Kalman Filter," EECE 5550 Mobile Robotics, Northeastern University, Oct. 2025.

[2] R. Zhang, M. Abdul Rahman, I. Premkumar, "ROS2 Cross-Platform Bridge Implementation," GitHub Repository, 2025. [Online]. Available: https://github.com/Jerryzhang258/EECE5550-Final-Project/tree/main/ros2%20crossplatform%20bridge