Final Project Report
Multi-point Navigation with Hand-Gesture Control

Submit to

Asst. Prof. Surat Kwanmuang, Ph.D.

Submitted By

Nara	Cheyklin	6538097021
Pannawish	Leechasan	6538114021
Pasin	Chanaphan	6538121321
Krittin	Kitjaruwannakkul	6538007521

1/2025: 2147427 Advanced Mobile Robot
Robotics and Artificial Intelligence Engineering (International Program) International School of Engineering (ISE) Faculty of Engineering, Chulalongkorn University

Introduction

Mobile robots play an increasingly important role in modern robotics due to their ability to navigate and interact with dynamic environments autonomously. Equipped with onboard sensors, computation, and actuators, mobile robots are widely used in applications such as logistics, service robotics, exploration, and human assistance. Effective operation of a mobile robot requires the integration of perception, localization, mapping, planning, and control, allowing the robot to understand its surroundings and make informed movement decisions. As these systems are often deployed in environments shared with humans, intuitive and flexible interaction methods are essential to ensure safety, usability, and adaptability. Consequently, research in mobile robotics increasingly focuses not only on autonomy but also on designing interaction mechanisms that enable humans to guide, supervise, and collaborate with robots in a natural and efficient manner.

This project focuses on a human–robot interaction system that integrates hand-gesture control with autonomous navigation on a TurtleBot3 using ROS 2 and the Nav2 framework. The goal is to move beyond traditional keyboard- or joystick-based control by enabling users to command robot behavior in a more natural and intuitive way. Instead of directly controlling motion, users specify navigation goals visually in RViz and use simple hand gestures to control when and how the robot executes those goals.

The system is designed as a distributed architecture that separates perception, interaction, and navigation according to computational needs. A USB camera mounted on the robot streams compressed image data over ROS, while MediaPipe-based hand-gesture recognition runs on a remote machine with greater processing capability. Gesture commands such as start, pause, continue, and stop are published as lightweight ROS topics, allowing real-time human intent to influence robot behavior without interfering with low-level navigation processes.

At the navigation level, a dedicated waypoint management node collects user-selected waypoints from RViz and interfaces with Nav2 through its action server to execute them sequentially. By combining gesture-based intent with autonomous planning and control, the project demonstrates a modular, scalable approach to human-in-the-loop robot navigation. This approach highlights how ROS 2 can support intuitive interaction while maintaining the robustness and structure required for real-world robotic systems.

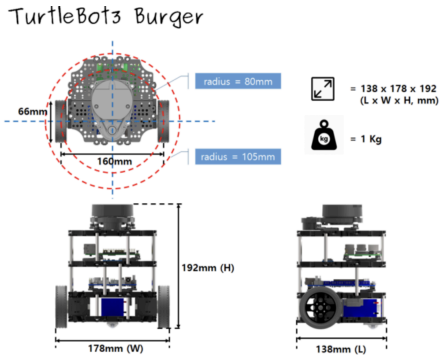


Fig1: Turtlebot3 Burger

Hardware Architecture

The TurtleBot3 Burger was chosen for its modular design, low cost, and strong support within the ROS community. Its hardware architecture includes a distributed set of components responsible for sensing, computation, and actuation. Together, these components form the foundation required for real-time navigation and human–robot interaction.

OpenCR Control Board

The OpenCR board serves as the robot's primary microcontroller, handling motor control, IMU processing, and low-level sensor data. Powered by a 32-bit STM32F7 chip, it executes control loops efficiently while communicating with the Raspberry Pi via USB. It also links to the Dynamixel motors through a TTL bus and provides stabilized power distribution, ensuring reliable operation during navigation.

Raspberry Pi 4 (On-Board Computer)

The Raspberry Pi 4 functions as the main processing unit, running ROS2, SLAM, navigation, and LiDAR handling. Its quad-core ARM CPU and 4GB RAM allow multiple ROS2 nodes to run simultaneously with real-time responsiveness. Ubuntu Server 20.04 provides a stable and lightweight platform for autonomous robot operation without relying on external computing.

LDS-02 LIDAR

The LDS-02 is the robot's primary 2D scanning sensor, producing 360° distance measurements at high frequency. With a range of up to 12 meters and strong signal stability, it provides reliable data for SLAM and localization. The sensor publishes scans through /scan, forming the basis of mapping accuracy and navigation performance.

DYNAMIXEL XL430-W250 Motor

The TurtleBot3 uses two XL430-W250 smart actuators in a differential-drive configuration. These motors offer precise velocity control and provide feedback such as torque, speed, and temperature. Their communication via a daisy-chained TTL bus simplifies wiring and supports responsive, accurate motion required for SLAM and navigation tasks.

USB Web Camera

In this project, a basic USB webcam was integrated into the TurtleBot3 as the primary vision sensor for real-time hand-gesture recognition. The webcam continuously captures live video of the user's hand and streams it to the processing workstation, where MediaPipe's hand-tracking model detects gestures and interprets them as navigation commands.

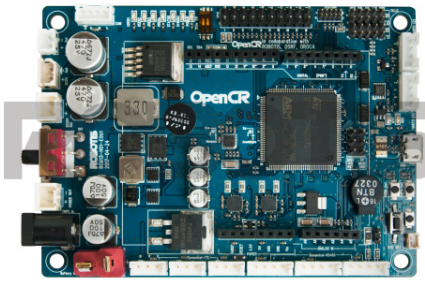


Fig2: OpenCR Control Board

Fig3: Raspberry Pi 4

Fig4: LDS-02 LIDAR

Fig5: DYNAMIXEL XL430-W250 Motor

Fig6: USB Web Camera

Software Architecture

The software stack for this project integrates ROS2 middleware, SLAM algorithms, navigation frameworks, and computer-vision components into a unified robotic system. Each layer contributes specific capabilities necessary for autonomy and interaction.

Ubuntu Linux and ROS2 Environment

Both the workstation PC and Raspberry Pi run Ubuntu to ensure compatibility with the ROS2 Humble distribution. ROS2 provides a modular communication framework using the Data Distribution Service (DDS), enabling reliable and low-latency exchange of messages between nodes. The system makes use of ROS2's composability, allowing multiple nodes, such as those handling sensor streams, SLAM, navigation, and teleoperation, to run concurrently. ROS2 also supports real-time constraints through executors and provides essential developer tools such as RViz2 for visualization and Gazebo for optional simulation environments.

SLAM Framework

SLAM is a foundational capability allowing the robot to autonomously build a map while estimating its position within that map. This project utilizes **Cartographer**, a graph-based SLAM algorithm that combines scan matching with real-time optimization. LDS-02 LiDAR data and wheel odometry are continuously fused to refine the robot's pose estimate. Cartographer generates a high-resolution 2D occupancy grid, which becomes the basis for navigation tasks. The reliability of the SLAM output directly affects path planning performance; therefore, stable LiDAR input and consistent motion during mapping are essential.

Navigation2 Stack

The Navigation2 (Nav2) framework enables the robot to plan and execute paths through its environment. Nav2 uses the AMCL algorithm to estimate the robot's pose on a pre-built map by comparing incoming LiDAR scans with map features. Once localization is established, the global planner generates an optimal path based on metrics such as Euclidean distance or costmap inflation. The local planner continuously adjusts the robot's velocity commands to avoid dynamic obstacles while following the global trajectory. Nav2's layered costmap system ensures that obstacles detected in real time immediately influence the robot's motion, preventing collisions.

MediaPipe Gesture-Recognition System

The interactive component of the system uses MediaPipe to detect, track, and classify hand gestures. MediaPipe's hand-tracking pipeline identifies 21 keypoints on the hand, enabling accurate representation of finger and palm orientation. A custom-developed Python module interprets these keypoints and classifies gestures such as an open palm, closed fist, or left/right tilt. These gestures are then translated into ROS2 Twist messages, allowing gesture-based teleoperation. The decision to integrate MediaPipe provides a modern, non-contact interaction method that increases accessibility and enhances the robot's usability in human-centered applications.

Fig7: Ubuntu 22.04 Desktop

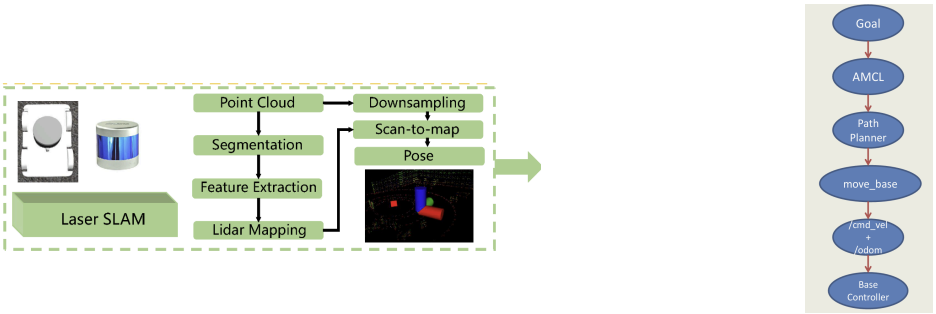


Fig8: SLAM and Navigation2 Framework

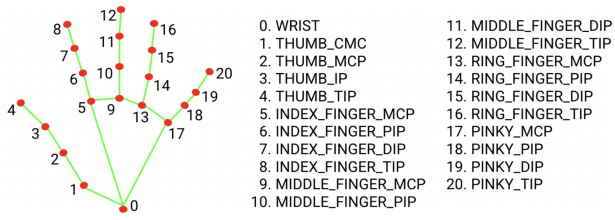


Fig9: MediaPipe Hand-Gesture Recognition

System Setup and Integration

PC and Network Configuration

The external workstation PC is configured to communicate with the TurtleBot3 over a shared local network. After installing ROS2 and TurtleBot3 packages, environment variables such as the robot model and ROS domain ID are set to ensure consistent communication. The PC primarily serves as a visualization and monitoring device using RViz2, while the Raspberry Pi conducts most of the computation.

Raspberry Pi Setup and Firmware Installation

The Raspberry Pi undergoes a structured installation process beginning with flashing Ubuntu Server, installing ROS2 Humble, and configuring TurtleBot3 packages. OpenCR firmware is updated using the official Robotis tools, enabling stable communication between the Raspberry Pi and motor system. Wi-Fi configuration ensures that the Pi and PC operate within the same ROS2 domain, allowing seamless communication through ROS topics and services.

SLAM Implementation

The SLAM process begins by launching Cartographer on the Raspberry Pi. As the robot is teleoperated around the environment, the LDS-02 sensor captures continuous scan data. Cartographer performs real-time scan matching, aligning new scans with previous ones to incrementally refine the map. The robot's wheel odometry supplements the scan matching process, reducing pose drift. Once the mapping is completed, the map is saved as a 2D occupancy grid, which becomes the navigation environment. The quality of the map is influenced by the robot's movement speed, sensor stability, and coverage of the environment, making controlled teleoperation essential.

Autonomous Navigation

After generating the map, the Nav2 stack is launched. The AMCL algorithm initializes the robot's position through an initial pose estimate selected in RViz2. AMCL continuously refines the pose estimate by comparing incoming LiDAR scans with known map features. Once localization stabilizes, users can assign target goals in RViz2. Nav2 generates a global path and computes real-time velocity commands for the robot to follow. The local planner dynamically adjusts to obstacles, ensuring safe navigation. The integration of costmaps, planners, and controllers allows the robot to move autonomously through the environment with minimal user intervention.

Implementation

After completing the system setup and basic integration in accordance with the TurtleBot3 e-Manual, the implementation of multi-point navigation with hand-gesture control using MediaPipe was carried out. The overall implementation consists of map creation, software dependency installation, node development, and system initialization across the robot and the remote computer.

Map Creation

A demonstration environment was prepared in a real-world setting, and SLAM was executed to generate a virtual occupancy grid map. The resulting map was saved as jjjmap.pgm and jjjmap.yaml, which were later used by the navigation stack for localization and path planning.

Installation of Required Packages

To support hand-gesture recognition, pip3 was first installed on the remote PC, followed by the installation of the MediaPipe library using pip3. The CVBridge package was installed via apt to enable image message conversion between ROS and OpenCV. On the TurtleBot3 Raspberry Pi, the v4l2-camera package was installed to interface with the USB webcam and publish image data to ROS topics.

Hardware Integration

A USB webcam was mounted at the front of the TurtleBot3 to capture hand gestures from the user. The camera continuously streams image data, which serves as input for MediaPipe-based gesture detection. Then, we incremented the radius of the turtlebot in Burger.yaml file by 0.07, or 7 centimeter.

Workspace and Node Development

A new ROS 2 workspace named nav_ws was created and built using colcon. Within the nav_ws/src directory, a new package named multi_point_nav was created. Two primary nodes were implemented within this package: hand_gesture_node, responsible for processing camera input and recognizing hand gestures, and multi_point_nav_node, which manages waypoint navigation by interfacing with the Nav2 action server.

Configuration of setup.py and package.xml

The setup.py file was edited to define executable entry points for both nodes, enabling them to be launched as ROS 2 executables. Additionally, the required runtime dependencies, including rclpy, geometry_msgs, nav2_msgs, std_msgs, sensor_msgs, and cv_bridge, were declared in the package.xml file to ensure proper dependency resolution.

Launch File Creation

To simplify system execution, a dedicated launch directory was created within the multi_point_nav package. A launch file named gesture_waypoint.launch.py was implemented to simultaneously start the hand gesture recognition node and the multi-point navigation node using a single command, thereby improving usability and repeatability.

Final Setup and System Initialization

On the TurtleBot3 Raspberry Pi, the robot bringup was initialized by running robot.launch.py. In a separate terminal on the Raspberry Pi, the USB webcam was activated using the v4l2_camera node with appropriate parameters for device selection, image resolution, and pixel format. Finally, on the remote PC, the complete project workflow was launched using gesture_waypoint.launch.py, enabling gesture-controlled multi-point navigation within the mapped environment.

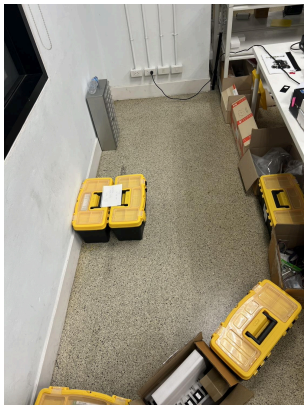


Fig10: Map in real world

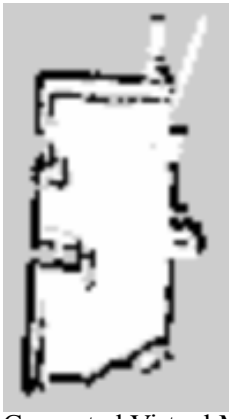
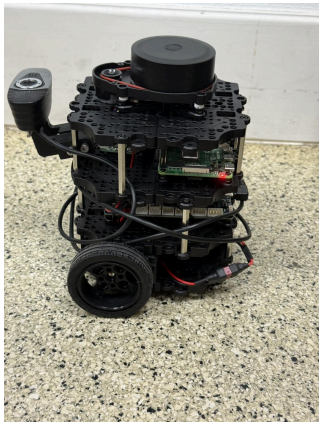


Fig11: Generated Virtual Map



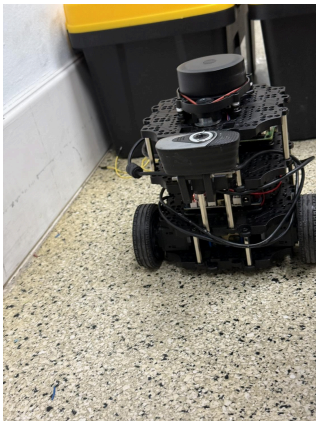
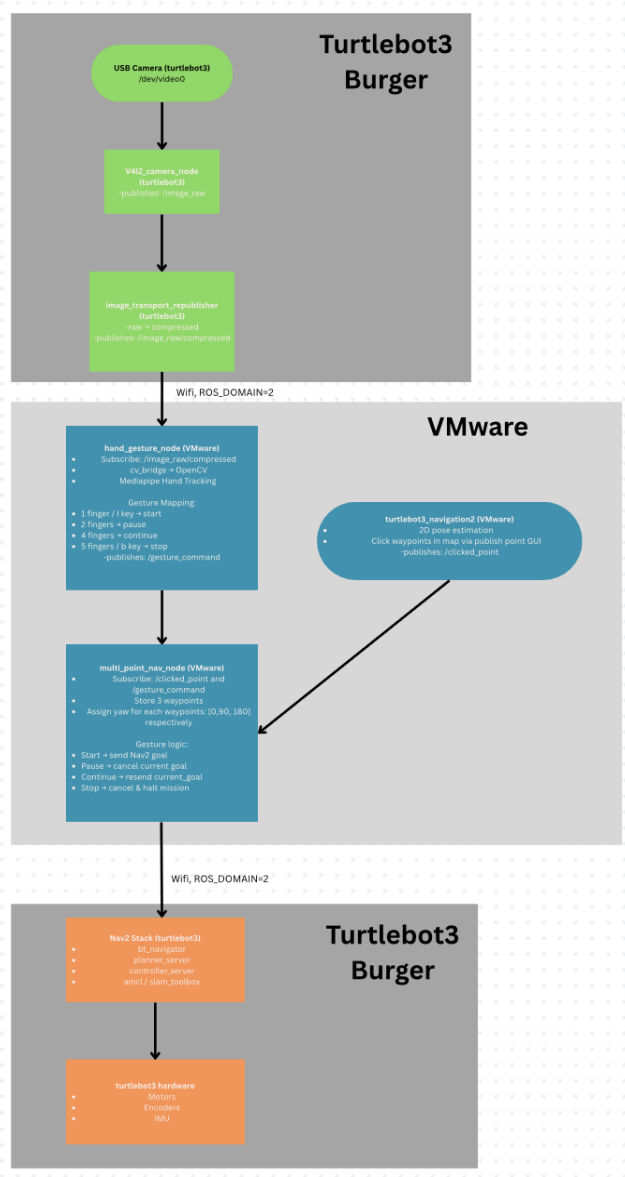


Fig12-13-14: Turtlebot mounted with Web Camera

Main Workflow



The user selects three waypoints in RViz using the Publish Point tool, which are collected by the multi_point_nav_nodeand assigned orientations. Meanwhile, a USB camera on the robot streams compressed images to the VMware machine, where the hand_gesture_node uses MediaPipe to recognize hand gestures and publish navigation commands. The multi_point_nav_node combines the waypoints and gesture commands to control when navigation starts, pauses, resumes, or stops, and sends goals to the Nav2 stack. Nav2 then plans and executes the robot's motion, driving the TurtleBot3 to each waypoint accordingly. (link to view this diagram in detail: https://www.canva.com/design/DAG7Z5LIKaw/rfdba5lFD8ZcnUoNwvREyg/edit?utm_content=DAG7Z5LIKaw&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton)

Source Code

hand_gesture_node.py

hand_gesture_node is a ROS 2 Python node that subscribes to the robot's camera stream on /image_raw/compressed, decodes each frame with cv_bridge, and runs MediaPipe Hands to detect a single hand and robustly count raised fingers regardless of hand orientation; it then maps specific finger counts to high-level navigation commands (start, pause, continue, stop) and publishes them as strings on /gesture_command, with optional keyboard overrides for testing, providing a lightweight, network-friendly human–robot interface that lets a user control Nav2 waypoint execution using simple hand gestures seen by the robot's camera.

```
import cv2
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

from sensor_msgs.msg import CompressedImage
from cv_bridge import CvBridge

import mediapipe as mp


class HandGestureNode(Node):
  def __init__(self):
    super().__init__('hand_gesture_node')

    self.pub = self.create_publisher(String, '/gesture_command', 10)
    self.bridge = CvBridge()

    self.sub = self.create_subscription(
      CompressedImage,
      '/image_raw/compressed',
      self.image_callback,
      10
    )

    self.mp_hands = mp.solutions.hands
    self.hands = self.mp_hands.Hands(
      max_num_hands=1,
      model_complexity=0,
      min_detection_confidence=0.5,
      min_tracking_confidence=0.5
    )
    self.mp_draw = mp.solutions.drawing_utils

    self.last_command = ""

    self.get_logger().info("HandGestureNode listening on /image_raw/compressed")

  def image_callback(self, msg: CompressedImage):
    frame = self.bridge.compressed_imgmsg_to_cv2(msg, desired_encoding='bgr8')
    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    results = self.hands.process(frame_rgb)
```

```
    command = ""

    if results.multi_hand_landmarks:
        hand = results.multi_hand_landmarks[0]
        finger_count = self.count_fingers(hand, frame.shape)

        if finger_count == 1:
            command = "start"
        elif finger_count == 2:
            command = "pause"
        elif finger_count == 4:
            command = "continue"
        elif finger_count == 5:
            command = "stop"

        self.mp_draw.draw_landmarks(frame, hand, self.mp_hands.HAND_CONNECTIONS)
        cv2.putText(frame, f"Fingers: {finger_count}", (10, 40),
                    cv2.FONT_HERSHEY_SIMPLEX, 1.0, (0, 255, 0), 2)

    # Keyboard overrides in VMware window
    key = cv2.waitKey(1) & 0xFF
    if key == ord('i'):
        command = "start"
    elif key == ord('b'):
        command = "stop"
    elif key == ord('q'):
        rclpy.shutdown()
        return

    if command and command != self.last_command:
        out = String()
        out.data = command
        self.pub.publish(out)
        self.get_logger().info(f"Published: {command}")
        self.last_command = command
    elif not command:
        self.last_command = ""

    cv2.imshow("Gesture (robot camera stream)", frame)

def count_fingers(self, hand_landmarks, img_shape):
"""
Robust finger counting:
- Detects whether the hand is oriented 'upright' or 'upside-down' in the image.
- Uses that to decide whether tip.y < pip.y means 'up' or the opposite.
- Thumb uses left/right hand detection if available; otherwise fallback.
"""
    h, w, _ = img_shape
    lm = hand_landmarks.landmark

    def px(i):
        return (lm[i].x * w, lm[i].y * h)

    # Key landmarks
    wrist_y = px(0)[1]
    middle_mcp_y = px(9)[1]   # middle finger MCP

    # If wrist is lower (bigger y) than MCP, fingers likely point upward in image
    # (because y increases downward)
```

```
        hand_upright = wrist_y > middle_mcp_y

        # Choose comparator for "finger is up"
        # Upright: tip above pip => tip_y < pip_y
        # Upside-down: tip below pip => tip_y > pip_y
        def finger_up(tip_id, pip_id):
            tip_y = px(tip_id)[1]
            pip_y = px(pip_id)[1]
            return tip_y < pip_y if hand_upright else tip_y > pip_y

        count = 0

        # Index, Middle, Ring, Pinky
        if finger_up(8, 6):   count += 1
        if finger_up(12, 10): count += 1
        if finger_up(16, 14): count += 1
        if finger_up(20, 18): count += 1

    # Thumb: tricky because it's sideways. Use x-axis, but direction depends on hand.
    # We'll infer hand side by comparing index MCP (5) vs pinky MCP (17):
    # If index_mcp_x < pinky_mcp_x => likely right hand in a normal (non-mirrored) view.
        index_mcp_x = px(5)[0]
        pinky_mcp_x = px(17)[0]
        right_hand_like = index_mcp_x < pinky_mcp_x

        thumb_tip_x = px(4)[0]
        thumb_ip_x  = px(3)[0]

    # If hand is mirrored, right_hand_like may flip; but this heuristic works surprisingly well.
        if right_hand_like:
            thumb_up = thumb_tip_x > thumb_ip_x
        else:
            thumb_up = thumb_tip_x < thumb_ip_x

        count += 1 if thumb_up else 0

        return count


def main():
    rclpy.init()
    node = HandGestureNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        cv2.destroyAllWindows()
        node.destroy_node()
        rclpy.shutdown()


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

multi_point_nav.py

multi_point_nav_node is a ROS 2 Python node that acts as a waypoint and mission controller for Nav2: it listens to RViz's /clicked_point topic to collect exactly three user-selected waypoints in the global map frame, converts them into PoseStamped goals with predefined yaw orientations, and interfaces with the Nav2 NavigateToPose action server to send these goals sequentially; at the same time, it subscribes to /gesture_command to start, pause, continue, or stop the mission based on hand-gesture or keyboard inputs, managing goal cancellation and resumption so that the robot can execute, interrupt, and safely resume a multi-waypoint navigation task under human-in-the-loop control.

```
import math

import rclpy

from rclpy.node import Node

from rclpy.action import ActionClient

from geometry_msgs.msg import PoseStamped, PointStamped

from std_msgs.msg import String

from nav2_msgs.action import NavigateToPose

def yaw_to_quaternion(yaw: float):

  qz = math.sin(yaw / 2.0)

  qw = math.cos(yaw / 2.0)

  return (0.0, 0.0, qz, qw)

class MultiPointNav(Node):

  def __init__(self):

    super().__init__('multi_point_nav')

    self._client = ActionClient(self, NavigateToPose, 'navigate_to_pose')

    self._sub_points = self.create_subscription(

      PointStamped, '/clicked_point', self.clicked_point_callback, 10

    )

    self._sub_cmd = self.create_subscription(

      String, '/gesture_command', self.command_callback, 10

    )

    self.points = []
```

```
        self.goal_poses = []
        self.current_goal_idx = 0
        self.goal_handle = None
        self.goals_prepared = False
        self.started = False
        self.paused = False
        self.stopped = False
        # ✅ EDIT yaw for each of the 3 waypoints here (degrees)
        yaw_degrees = [0.0, 90.0, 180.0]
        self.yaws = [math.radians(y) for y in yaw_degrees]
        self.get_logger().info(
            "MultiPointNav ready.\n"
            "- Click 3 points in RViz using 'Publish Point'\n"
            "- Commands: start(1 finger or i), pause(2), continue(4), stop(5 or b)"
        )
    # ---------- RViz clicks ----------
    def clicked_point_callback(self, msg: PointStamped):
        if self.goals_prepared:
            self.get_logger().warn("Already have 3 points; ignoring extra clicks.")
            return
        self.points.append(msg)
        self.get_logger().info(
            f"Collected point {len(self.points)}: ({msg.point.x:.2f}, {msg.point.y:.2f}) frame={msg.header.frame_id}"
        )
        if len(self.points) == 3:
            self.prepare_goals()
    def prepare_goals(self):
        self.get_logger().info("Waiting for Nav2 action server 'navigate_to_pose'...")
```

```
    if not self._client.wait_for_server(timeout_sec=10.0):
        self.get_logger().error("navigate_to_pose action server not available.")
        return
    self.goal_poses = []
    for i, pt in enumerate(self.points):
        pose = PoseStamped()
        pose.header.frame_id = pt.header.frame_id  # usually 'map'
        pose.header.stamp = self.get_clock().now().to_msg()
        pose.pose.position.x = pt.point.x
        pose.pose.position.y = pt.point.y
        pose.pose.position.z = 0.0

        yaw = self.yaws[i] if i < len(self.yaws) else self.yaws[-1]
        qx, qy, qz, qw = yaw_to_quaternion(yaw)
        pose.pose.orientation.x = qx
        pose.pose.orientation.y = qy
        pose.pose.orientation.z = qz
        pose.pose.orientation.w = qw
        self.goal_poses.append(pose)

        self.get_logger().info(
            f"Prepared goal {i+1}: ({pose.pose.position.x:.2f}, {pose.pose.position.y:.2f}), yaw={yaw:.2f} rad"
        )
    self.goals_prepared = True
    self.started = False
    self.paused = False
    self.stopped = False
    self.current_goal_idx = 0
```

```
    self.goal_handle = None



    self.get_logger().info(

        "✅ Goals prepared.\n"

        "Show 1 finger or press 'i' in gesture window to START.\n"

        "2 fingers PAUSE, 4 fingers CONTINUE, 5 fingers or 'b' STOP."

    )

# ---------- Commands ----------

def command_callback(self, msg: String):

    cmd = msg.data.strip().lower()

    self.get_logger().info(f"Command: {cmd}")

    if cmd == "start":

        if not self.goals_prepared:

            self.get_logger().warn("Start ignored: no goals yet (click 3 points first).")

            return

        self.stopped = False

        self.paused = False

        self.started = True

        if self.goal_handle is None:

            self.send_current_goal()

    elif cmd == "pause":

        if self.goal_handle is None:

            self.get_logger().info("Pause ignored: no active goal.")

            return

        if self.paused:

            self.get_logger().info("Already paused.")

            return

        self.paused = True
```

```
            self.started = False
            self.get_logger().info("Pausing: canceling current goal.")
            self.goal_handle.cancel_goal_async().add_done_callback(lambda f: self._after_cancel("pause", f))
        elif cmd == "continue":
            if not self.goals_prepared:
                self.get_logger().warn("Continue ignored: no goals yet.")
                return
            if not self.paused:
                self.get_logger().info("Continue ignored: not paused.")
                return
            self.paused = False
            self.stopped = False
            self.started = True
            if self.goal_handle is None:
                self.get_logger().info("Continuing: resending current goal.")
                self.send_current_goal()
        elif cmd == "stop":
            self.stopped = True
            self.started = False
            self.paused = False
            if self.goal_handle is not None:
                self.get_logger().info("Stopping: canceling current goal.")
                self.goal_handle.cancel_goal_async().add_done_callback(lambda f: self._after_cancel("stop", f))
            else:
                self.get_logger().info("Stopped (no active goal).")
    # ---------- Nav2 goal sending ----------
    def send_current_goal(self):
        if not self.goals_prepared:
```

```
            return

        if self.stopped or self.paused or not self.started:

            return

        if self.current_goal_idx >= len(self.goal_poses):

            self.get_logger().info("🎉 All goals completed.")

            return

        pose = self.goal_poses[self.current_goal_idx]

        goal = NavigateToPose.Goal()

        goal.pose = pose

        self.get_logger().info(

            f"Sending goal {self.current_goal_idx+1}/{len(self.goal_poses)}: "

            f"({pose.pose.position.x:.2f}, {pose.pose.position.y:.2f})"

        )

        self._client.send_goal_async(goal).add_done_callback(self._goal_response_cb)

    def _goal_response_cb(self, future):

        gh = future.result()

        if not gh.accepted:

            self.get_logger().warn(f"Goal {self.current_goal_idx+1} rejected. Skipping.")

            self.goal_handle = None

            self.current_goal_idx += 1

            self.send_current_goal()

            return

        self.goal_handle = gh

        self.get_logger().info(f"Goal {self.current_goal_idx+1} accepted.")

        gh.get_result_async().add_done_callback(self._result_cb)



    def _result_cb(self, future):

        _ = future.result().result
```

```
        self.get_logger().info(f"Goal {self.current_goal_idx+1} finished.")
        self.goal_handle = None

        if self.stopped or self.paused or not self.started:
            self.get_logger().info("Not running; will not send next goal.")
            return
        self.current_goal_idx += 1
        self.send_current_goal()
    def _after_cancel(self, reason: str, future):
        try:
            _ = future.result()
        except Exception as e:
            self.get_logger().error(f"Cancel error ({reason}): {e}")
        self.goal_handle = None
        if reason == "pause":
            self.get_logger().info("Paused. Show 4 fingers to CONTINUE.")
        elif reason == "stop":
            self.get_logger().info("Stopped.")
def main():
    rclpy.init()
    node = MultiPointNav()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()
```

```
if __name__ == "__main__":

    main()
```

gesture_waypoint.launch.py

gesture_waypoint.launch is a ROS launch file that combines commands to run three different nodes accordingly into a single file, which includes

- ros2 launch turtlebot3_navigation2 navigation2.launch.py map:=$HOME/jjjmap.yaml
- ros2 run multi_point_nav multi_point_nav_node
- ros2 run multi_point_nav hand_gesture_node

```
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory
import os


def generate_launch_description():

    # Path to turtlebot3_navigation2 launch file
    tb3_nav2_launch = os.path.join(
        get_package_share_directory('turtlebot3_navigation2'),
        'launch',
        'navigation2.launch.py'
    )

    # Map file (edit if needed)
    map_yaml = os.path.expanduser('~/jjjmap.yaml')

    return LaunchDescription([

        # ———————————— Nav2 ————————————
        IncludeLaunchDescription(
            PythonLaunchDescriptionSource(tb3_nav2_launch),
            launch_arguments={
                'map': map_yaml
            }.items()
        ),

        # ———————————— Waypoint logic ————————————
        Node(
            package='multi_point_nav',
            executable='multi_point_nav_node',
            name='multi_point_nav_node',
            output='screen',
        ),

        # ———————————— Mediapipe gesture node ————————————
        Node(
```

```
                package='multi_point_nav',
                executable='hand_gesture_node',
        name='hand_gesture_node',
        output='screen',
    ),
])
```

Result

Discussion

One of the most significant challenges encountered during the development process arose during initial testing of Nav2 goal execution in RViz. When issuing basic navigation goals, the TurtleBot consistently moved in the opposite direction from the intended target. This issue required extensive debugging, as initial assumptions focused on potential problems with the LiDAR sensor or the OpenCR controller. After thorough investigation, the root cause was identified as an incorrect motor placement on the robot chassis. The issue was resolved by disassembling the TurtleBot, swapping the motor connections, and reassembling the system, after which the robot exhibited correct directional behavior.

Another challenge was observed during navigation near walls in the generated map. When the TurtleBot faced a wall that contained small gaps or discontinuities, the robot occasionally attempted to pass through these openings, leading to unsafe or unintended trajectories. This behavior was mitigated by tuning parameters within the burger.yaml configuration file, particularly those related to the local and global costmaps. By adjusting obstacle inflation and costmap resolution parameters, the navigation stack was able to more accurately represent environmental boundaries and avoid narrow discontinuities.

Outcome

During the final system demonstration, the integrated vision and navigation pipeline operated reliably. The use of the /image_raw/compressed topic enabled smooth video streaming with reduced network latency, allowing MediaPipe to process hand gestures in real time. Accurate gesture recognition was achieved when the user’s hand approached the camera from behind the TurtleBot, a requirement imposed by the camera’s coordinate frame orientation, where the y-axis is aligned from the rear to the front of the robot. This orientation was found to significantly influence the finger-counting logic implemented in the hand_gesture_node.

When all operational conditions were satisfied, the TurtleBot successfully navigated to waypoints selected in RViz and responded correctly to gesture-based commands. These commands could dynamically override ongoing navigation tasks and were further complemented by keyboard-based control, demonstrating a flexible and robust human-in-the-loop navigation system.

Video Demonstration

https://youtu.be/Uw6x6yV6_6k?si=UW3o-mwzbli-Bdvc

Further Implementation

1. Enhanced Interaction Layer

- Expand beyond fixed finger-count commands to a small command vocabulary with confirmation/feedback
- Add gestures/commands: "confirm" (with keyboard fallback), "reset waypoints", "return home" (to stored docking pose)
- Visual RViz feedback: Publish MarkerArray for waypoints, yaw arrows, active target, mission state (RUN/PAUSE/STOP)

2. Robust Gesture Perception

- Upgrade finger-count heuristic → MediaPipe handedness + palm orientation
- Add temporal smoothing (majority vote over N frames) + confidence thresholds to prevent flicker
- Lightweight calibration for user-specific gestures
- Auto-adapt to camera placement (mirrored/rotated detection)
- WiFi reliability: Adaptive streaming (resolution/FPS throttling, compressed quality) + health monitoring
- Fallback: Disable gestures → keyboard-only on packet loss/latency spikes + user warnings

3. Advanced Navigation & Safety

- Generalize from "three points" → arbitrary waypoints, looping patrols, dynamic reordering, conditional execution
- Deeper Nav2 behavior tree integration: "navigate → rotate → wait gesture → proceed", "navigate → snapshot → publish"
- Safety features:
    - Speed limiting during manual override
    - Auto-pause on high obstacle proximity

Reference

[1] ROBOTIS, “TurtleBot3 e-Manual,” ROBOTIS Co., Ltd. [Online]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/. Accessed: Sep. 2025.4[2] ROS 2 Documentation, “ROS 2: Robot Operating System,” Open Robotics. [Online]. Available: https://docs.ros.org/en/humble/. Accessed: Sep. 2025.

[3] S. Macenski, T. Moore, D. Lu, A. Merzlyakov, and M. Ferguson, “Navigation2: A modern navigation framework for ROS 2,” *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[4] Google, “MediaPipe: A Framework for Building Perception Pipelines,” Google Research. [Online]. Available: https://mediapipe.dev/. Accessed: Sep. 2025.

[5] M. J. Cheok, Z. Omar, and M. H. Jaward, “A review of hand gesture and sign language recognition techniques,” *International Journal of Machine Learning and Cybernetics*, vol. 10, no. 1, pp. 131–153, 2019.

[6] Open Robotics, “cv_bridge: ROS–OpenCV Interface,” ROS Wiki. [Online]. Available: https://wiki.ros.org/cv_bridge. Accessed: Sep. 2025.

[7] Open Robotics, “image_transport: Efficient image transport in ROS,” ROS Wiki. [Online]. Available: https://wiki.ros.org/image_transport. Accessed: Sep. 2025.

[8] Open Robotics, “Nav2: ROS 2 Navigation Stack,” Open Robotics. [Online]. Available: https://navigation.ros.org/. Accessed: Sep. 2025.