HandDetector Node: Comprehensive Technical Documentation with Theoretical Equations

Automated Documentation

May 13, 2025

Contents

1 Introduction

The `HandDetector` node is a sophisticated ROS 2 (Robot Operating System) component designed to enable human-robot interaction through real-time hand detection and 3D pose estimation. It leverages the MediaPipe Hands framework for detecting hand landmarks, integrates LiDAR depth measurements for 3D localization, and uses TF2 for coordinate frame transformations. This node is ideal for service robots in environments like retail, hospitality, or assistive robotics, where a raised hand signals a request for interaction.

This documentation provides an exhaustive guide for developers, detailing the node's architecture, algorithms, workflow, dependencies, theoretical equations, and usage. It includes a system architecture diagram, pseudocode for key algorithms, a dedicated section on equations with theoretical derivations, example use cases, performance considerations, and troubleshooting strategies. The document assumes familiarity with ROS 2, Python, computer vision, and robotics concepts.

2 Purpose and Objectives

The `HandDetector` node serves the following objectives:

- **Hand Detection**: Identifies raised hands in camera images using MediaPipe Hands, analyzing landmarks (e.g., wrist, fingertips).
- **3D Pose Estimation**: Computes the 3D position and orientation of a customer using 2D hand coordinates and LiDAR depth, transformed to the `map` frame.
- **Real-Time Processing**: Operates at a configurable rate (default 10 Hz) for timely responses.
- **Reliability**: Uses temporal filtering to reduce false positives/negatives and median filtering for stable depth measurements.
- **Debugging Support**: Publishes annotated debug images for visualization and troubleshooting.

2.1 Use Cases

- **Retail**: A robot detects a customer's raised hand to offer assistance.
- **Hospitality**: A restaurant robot responds to a raised hand for order-taking.
- **Assistive Robotics**: A healthcare robot aids patients signaling with a raised hand.

3 Dependencies

The node relies on:

- **ROS 2**: Framework for node management (Humble or later).
- **Python 3**: Programming language (3.8 or higher).
- **rclpy**: ROS 2 Python client library.
- **sensor_msgs**: `Image` and `LaserScan` message types.
- **geometry_msgs**: `PoseStamped` and `Quaternion` message types.
- **std_msgs**: `Bool` message type.

- **cv_bridge**: Converts ROS 2 `Image` messages to OpenCV images.
- **OpenCV (cv2)**: Image processing (4.5 or higher).
- **NumPy**: Numerical computations (1.21 or higher).
- **MediaPipe**: Hand detection (0.8.9 or higher).
- **tf2_ros**: Coordinate frame transformations.
- **tf2_geometry_msgs**: `PoseStamped` transformations.

3.1 Installation

Install dependencies:

```
sudo apt update
sudo apt install ros-<distro>-rclpy ros-<distro>-sensor-msgs ros-<
    distro>-geometry-msgs ros-<distro>-std-msgs ros-<distro>-cv-bridge
    ros-<distro>-tf2-ros ros-<distro>-tf2-geometry-msgs
pip install opencv-python==4.5.5.64 numpy==1.21.6 mediapipe==0.8.9.1
```

Replace `<distro>` with your ROS 2 distribution (e.g., `humble`).

4 Inputs and Outputs

4.1 Inputs

- **/image_raw** (`sensor_msgs/Image`):
  - Format: BGR8, typically 640x480 pixels, 30 FPS.
  - Purpose: Hand detection and 2D coordinate extraction.
- **/scan** (`sensor_msgs/LaserScan`):
  - Format: Angular depth measurements, 0.5-degree resolution.
  - Purpose: 3D pose estimation via depth data.

4.2 Outputs

- **/hand_raised** (`std_msgs/Bool`):
  - Format: `True` if hand raised (2 consecutive detections), `False` otherwise (3 consecutive non-detections).
  - Purpose: Signals gesture detection.
- **/customer_pose** (`geometry_msgs/PoseStamped`):
  - Format: 3D position (x, y, z) and quaternion orientation in `map` frame.
  - Purpose: Customer localization.
- **/hand_detection_debug** (`sensor_msgs/Image`):
  - Format: BGR8 image with landmarks and annotations.
  - Purpose: Visualizes detection results.

4.3 Parameters

Parameter	**Type**	**Default**	**Description**
`processing_rate`	Float	10.0	Processing frequency (Hz).
`downscale_factor`	Integer	2	Image downscaling factor.
`hand_raised_threshold`	Float	0.25	Normalized vertical distance for raised hand.
`detection_confidence`	Float	0.6	MediaPipe detection confidence.
`tracking_confidence`	Float	0.5	MediaPipe tracking confidence.
`model_complexity`	Integer	0	MediaPipe model complexity (0 = lite, 1 = full).
`camera_fx`, `camera_fy`	Float	460.0	Camera focal lengths (pixels).
`camera_cx`, `camera_cy`	Float	320.0, 240.0	Camera principal point (pixels).

Table 1: Configurable parameters.

5 Theoretical Equations

The `HandDetector` node employs several mathematical equations for hand detection, pose estimation, and coordinate transformations. Below, each equation is presented with its theoretical basis and derivation.

5.1 Camera Angle Calculation

In `calculate_customer_pose`, the horizontal angle from the camera center to the hand is computed:

$$\theta = \arctan\left(\frac{p_x - c_x}{f_x}\right) \tag{1}$$

where:

- p_x: x-coordinate of the hand (wrist) in the downscaled image (pixels).
- c_x: Camera principal point x-coordinate (pixels, adjusted for downscaling).
- f_x: Camera focal length in x-direction (pixels, adjusted for downscaling).
- θ: Horizontal angle (radians).

Theoretical Basis: This equation derives from the pinhole camera model, where a point in the image plane is projected onto the 3D world. The angle θ represents the angular offset of the hand from the camera's optical axis in the x-direction. The tangent function relates the pixel offset $(p_x - c_x)$ to the focal length (f_x), assuming a flat image plane.

5.2 3D Position Calculation

The 3D position of the hand in the `camera_link` frame is computed:

$$x = d \cdot \cos(\theta) \tag{2}$$

$$y = d \cdot \sin(\theta) \tag{3}$$

$$z = 0 \tag{4}$$

where:

- d: Depth from LiDAR (meters).

- θ: Camera angle (radians).
- x, y, z: 3D coordinates in the camera frame (meters).

Theoretical Basis: These equations project the depth measurement along the angle θ to obtain the 3D position. The camera frame is assumed to have its origin at the camera, with the x-axis along the optical axis, y-axis horizontal, and z-axis vertical. The $z = 0$ assumption simplifies the model, assuming the hand is at the robot's height.

5.3 Orientation Calculation

The customer's orientation is computed as a quaternion facing the robot:

$$\phi = \arctan 2(y, x) + \pi \tag{5}$$

$$\mathbf{q} = \text{euler\_to\_quaternion}(0, 0, \phi) \tag{6}$$

where:

- x, y: Hand position in the camera frame.
- ϕ: Yaw angle (radians).
- $\mathbf{q}$: Quaternion representing orientation.

Theoretical Basis: The $\arctan 2(y, x)$ function computes the angle of the hand relative to the camera's x-axis. Adding π rotates the orientation 180 degrees, ensuring the customer faces the robot (opposite the hand-to-camera vector). The `euler_to_quaternion` function converts the yaw angle to a quaternion.

5.4 Euler to Quaternion Conversion

The `euler_to_quaternion` function converts Euler angles to a quaternion:

$$q_x = \sin\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) - \cos\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right) \tag{7}$$

$$q_y = \cos\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) + \sin\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right) \tag{8}$$

$$q_z = \cos\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right) - \sin\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) \tag{9}$$

$$q_w = \cos\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) + \sin\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right) \tag{10}$$

where:

- ϕ, θ, ψ: Roll, pitch, yaw angles (radians). In this case, $\phi = \theta = 0$, $\psi = \phi$.
- q_x, q_y, q_z, q_w: Quaternion components.

Theoretical Basis: This conversion follows the standard Euler-to-quaternion formula, using a ZYX rotation sequence. Since only yaw is non-zero, the equations simplify significantly, but the full form is implemented for generality.

5.5 LiDAR Index Calculation

The LiDAR scan index is computed:

$$\text{index} = \left\lfloor \frac{\theta - \theta_{\min}}{\Delta\theta} \right\rfloor \tag{11}$$

where:

- θ: Normalized angle (radians).
- θ_{min}: Minimum scan angle (radians).
- $\Delta\theta$: Angle increment (radians).
- index: Integer index into the scan array.

Theoretical Basis: This equation maps a continuous angle to a discrete index in the LiDAR scan array, accounting for the scan's angular resolution.

5.6 Angle Normalization

Angles are normalized to $[-\pi, \pi]$:

$$\theta_{\text{norm}} = \theta - 2\pi \cdot \left\lfloor \frac{\theta + \pi}{2\pi} \right\rfloor \tag{12}$$

where:

- θ: Input angle (radians).
- θ_{norm}: Normalized angle (radians).

Theoretical Basis: This ensures angles are within a consistent range, avoiding discontinuities in LiDAR index calculations.

6 System Architecture

The node integrates multiple components:

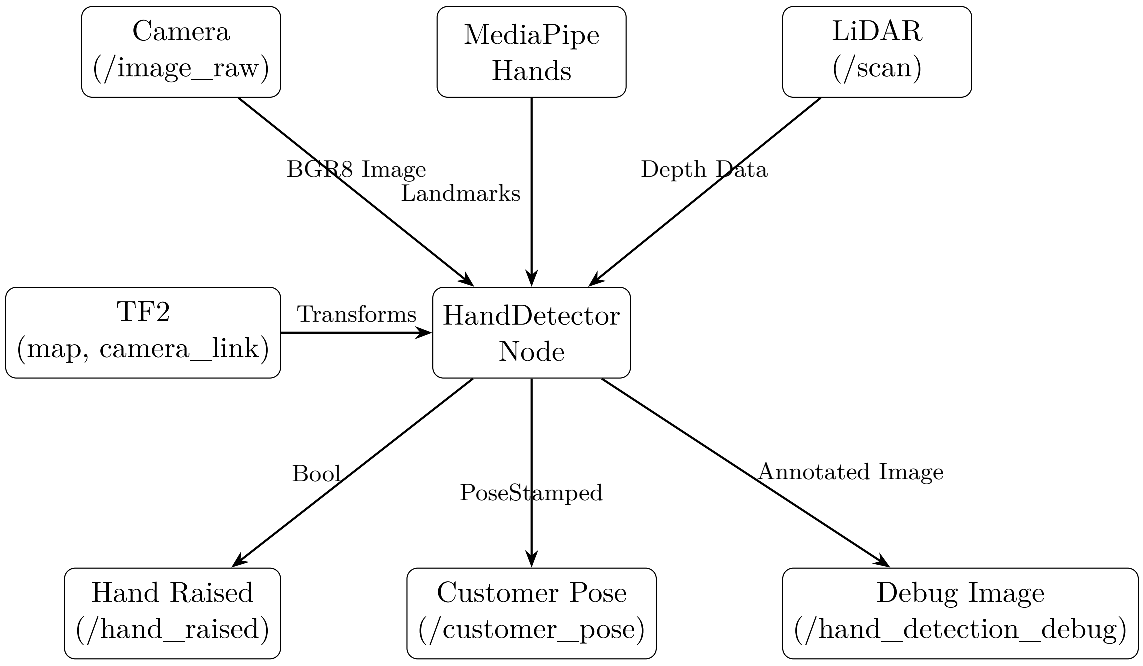

Figure 1: System architecture and data flow.

6.1 Component Details

- **Camera**: Provides BGR8 images at 30 FPS via `/image_raw`.
- **LiDAR**: Supplies depth scans via `/scan`.

- **MediaPipe Hands**: Detects hand landmarks.
- **TF2**: Manages frame transformations.
- **HandDetector Node**: Orchestrates processing and publishing.
- **Publishers**: Output hand status, pose, and debug images.

7 Detailed Workflow

7.1 Initialization

1. Create ROS 2 node (`hand`$_{d}etector).Declareandretrieveparameters.$
2. Initialize MediaPipe Hands.
3. Set up camera intrinsic matrix.
4. Initialize TF2 buffer and listener.
5. Subscribe to `/image_raw` and `/scan`.
6. Create publishers.
7. Initialize state variables.

7.2 Image Processing (`image_callback`)

7.3 LiDAR Processing (`lidar_callback`)

7.4 Hand Detection (`detect_hand_and_position`)

7.5 Pose Estimation (`calculate_customer_pose`)

7.6 Depth Estimation (`get_depth_from_lidar`)

8 Code Structure

- **___init___**: Initializes node, parameters, MediaPipe, camera matrix, TF2, subscriptions, and publishers.
- **image_callback**: Processes images, detects hands, filters temporally, estimates poses, and publishes.
- **lidar_callback**: Stores LiDAR data.
- **detect_hand_and_position**: Detects hands and generates debug images.
- **calculate_customer_pose**: Computes 3D poses.
- **get_depth_from_lidar**: Retrieves depth measurements.
- **euler_to_quaternion**: Converts Euler angles to quaternions.
- **get_latest_transform_time**: Retrieves transform timestamps.

9 Performance Considerations

9.1 Computational Load

- **MediaPipe**: Dominant bottleneck, scaling with image resolution and `model_complexity`.

Algorithm 1 Image Processing

```
1: $current\_time \leftarrow get\_clock().now()$
2: **if** $(current\_time - last\_process\_time) < min\_process\_interval$ **then**
3:     **return**
4: **end if**
5: $last\_process\_time \leftarrow current\_time$
6: $cv\_image \leftarrow bridge.imgmsg\_to\_cv2(msg,' bgr8')$
7: $processed\_image \leftarrow resize(cv\_image, 1/downscale\_factor)$
8: $hand\_raised, hand\_coords, debug\_image \leftarrow detect\_hand\_and\_position(processed\_image)$
9: **if** $hand\_raised$ **then**
10:     $hand\_raised\_streak \leftarrow hand\_raised\_streak + 1$
11:     $hand\_lowered\_streak \leftarrow 0$
12:     **if** $hand\_raised\_streak \geq 2$ **then**
13:         $is\_hand\_raised \leftarrow True$
14:         $last\_hand\_position \leftarrow hand\_coords$
15:     **end if**
16: **else**
17:     $hand\_lowered\_streak \leftarrow hand\_lowered\_streak + 1$
18:     $hand\_raised\_streak \leftarrow 0$
19:     **if** $hand\_lowered\_streak \geq 3$ **then**
20:         $is\_hand\_raised \leftarrow False$
21:     **end if**
22: **end if**
23: $publish(hand\_raised\_msg, is\_hand\_raised)$
24: **if** $is\_hand\_raised \wedge last\_hand\_position \neq None$ **then**
25:     $customer\_pose \leftarrow calculate\_customer\_pose(last\_hand\_position)$
26:     **if** $customer\_pose \neq None$ **then**
27:         $publish(customer\_pose)$
28:     **end if**
29: **end if**
30: $publish(debug\_image)$
```

Algorithm 2 LiDAR Callback

```
1: $latest\_scan \leftarrow msg$
2: $latest\_scan\_time \leftarrow msg.header.stamp$
```

Algorithm 3 Hand Detection

```
1: $image\_rgb \leftarrow cvtColor(image, BGR2RGB)$
2: $results \leftarrow hands.process(image\_rgb)$
3: $hand\_raised \leftarrow False$
4: $hand\_coords \leftarrow None$
5: $debug\_image \leftarrow image.copy()$
6: if $results.multi\_hand\_landmarks$ then
7:     for $hand\_landmarks \in results.multi\_hand\_landmarks$ do
8:         $wrist \leftarrow hand\_landmarks[WRIST]$
9:         $middle\_tip \leftarrow hand\_landmarks[MIDDLE\_FINGER\_TIP]$
10:        $thumb\_tip \leftarrow hand\_landmarks[THUMB\_TIP]$
11:        $pinky\_tip \leftarrow hand\_landmarks[PINKY\_TIP]$
12:        $vertical\_distance \leftarrow wrist.y - middle\_tip.y$
13:        $hand\_span \leftarrow |thumb\_tip.x - pinky\_tip.x|$
14:        if $vertical\_distance > hand\_raised\_threshold \land hand\_span > 0.15$ then
15:            $hand\_raised \leftarrow True$
16:            $px \leftarrow wrist.x \cdot width$
17:            $py \leftarrow wrist.y \cdot height$
18:            $hand\_coords \leftarrow (px, py)$
19:            $draw\_landmarks(debug\_image, hand\_landmarks)$
20:            $draw\_circle(debug\_image, hand\_coords)$
21:            $putText(debug\_image, 'HANDRAISED', vertical\_distance)$
22:        end if
23:    end for
24: else
25:    $putText(debug\_image, 'NOHANDDETECTED')$
26: end if
27: return $hand\_raised, hand\_coords, debug\_image$
```

Algorithm 4 Pose Estimation

```
1: $fx \leftarrow camera\_matrix[0, 0]/downscale\_factor$
2: $cx \leftarrow camera\_matrix[0, 2]/downscale\_factor$
3: $px, py \leftarrow hand\_coords$
4: $angle \leftarrow \arctan((px - cx)/fx)$
5: $depth \leftarrow get\_depth\_from\_lidar(angle)$
6: if $depth = None \lor \neg isfinite(depth)$ then
7:     return $None$
8: end if
9: $transform\_time \leftarrow get\_latest\_transform\_time()$
10: $pose \leftarrow PoseStamped()$
11: $pose.header.frame\_id \leftarrow 'camera\_link'$
12: $pose.header.stamp \leftarrow transform\_time$
13: $pose.position.x \leftarrow depth \cdot \cos(angle)$
14: $pose.position.y \leftarrow depth \cdot \sin(angle)$
15: $pose.position.z \leftarrow 0.0$
16: $angle\_to\_customer \leftarrow \arctan 2(pose.position.y, pose.position.x)$
17: $pose.orientation \leftarrow euler\_to\_quaternion(0, 0, angle\_to\_customer + \pi)$
18: $transform \leftarrow lookup\_transform('map', 'camera\_link', pose.header.stamp)$
19: $transformed\_pose \leftarrow do\_transform\_pose\_stamped(pose, transform)$
20: return $transformed\_pose$
```

Algorithm 5 Depth Estimation

```
1: if $latest\_scan = None$ then
2:     return $None$
3: end if
4: $angle \leftarrow normalize\_angle(angle\_rad)$
5: $index \leftarrow \lfloor (angle - latest\_scan.angle\_min)/latest\_scan.angle\_increment \rfloor$
6: $valid\_ranges \leftarrow []$
7: for $i \in [index - 3, index + 3]$ do
8:     if $latest\_scan.range\_min \leq ranges[i] \leq latest\_scan.range\_max$ then
9:         $valid\_ranges.append(ranges[i])$
10:    end if
11: end for
12: if $valid\_ranges \neq []$ then
13:     return $median(valid\_ranges)$
14: else
15:     return $None$
16: end if
```

- **Image Resizing**: Fast but dependent on input size.
- **LiDAR Processing**: Minimal due to selective sampling.
- **TF2**: Low overhead unless transforms are unavailable.

9.2 Real-Time Constraints

- **Processing Rate**: 10 Hz is typical, adjustable via `processing_rate`.
- **Latency**: 50-100 ms per frame on modern hardware.
- **Optimizations**: Increase `downscale_factor`, use `model_complexity=0`.

10 Error Handling

- **Image Errors**: Caught and logged in `image_callback`.
- **TF2 Failures**: Handled by returning `None` and logging warnings.
- **LiDAR Absence**: Returns `None` if no valid depths.
- **MediaPipe Failures**: Continues processing with debug annotations.

11 Usage Instructions

1. Install dependencies (Section 3).
2. Source ROS 2:

```bash
source /opt/ros/<distro>/setup.bash
source <your_workspace>/install/setup.bash
```

3. Launch node:

```bash
ros2 run <your_package_name> perception.py
```

4. Configure parameters:

```
ros2 run <your_package_name> perception.py --ros-args -p
    processing_rate:=5.0
```

5. Visualize outputs with `rviz2`, `ros2 topic echo`, and `rqt_image_view`.

12 Example Configuration

```
hand_detector:
  ros__parameters:
    processing_rate: 8.0
    downscale_factor: 3
    hand_raised_threshold: 0.3
    detection_confidence: 0.7
    tracking_confidence: 0.6
    model_complexity: 1
    camera_fx: 500.0
    camera_fy: 500.0
    camera_cx: 320.0
    camera_cy: 240.0
```

Load with:

```
ros2 run <your_package_name> perception.py --ros-args --params-file
    config.yaml
```

13 Troubleshooting Guide

- **No Hand Detection**: Verify `/image_raw`, lighting, and `detection_confidence`.
- **No Pose**: Check `/scan`, TF2 transforms, and `latest_scan`.
- **High CPU**: Increase `downscale_factor`, reduce `processing_rate`.
- **Debug Image Issues**: Confirm `/hand_detection_debug` topic and `CvBridge`.

14 Maintenance and Extension

14.1 Maintenance

- Monitor logs for errors.
- Tune parameters for environment.
- Calibrate camera intrinsics.

14.2 Extensions

- Multi-hand detection.
- Gesture recognition.
- Depth camera integration.
- Adaptive thresholds.

15 Limitations and Future Work

- **Limitations**: Single hand detection, LiDAR dependency, fixed camera intrinsics.
- **Future Work**: Multi-hand support, depth cameras, gesture recognition.

16 Conclusion

The `HandDetector` node enables robust hand detection and pose estimation for human-robot interaction. This documentation, with detailed equations, workflow, and practical guidance, equips developers to deploy and enhance the system.