ROS2 Sensor Calibration & Visualization Using TurtleBot3 (Simplified Training Version)

1. Introduction

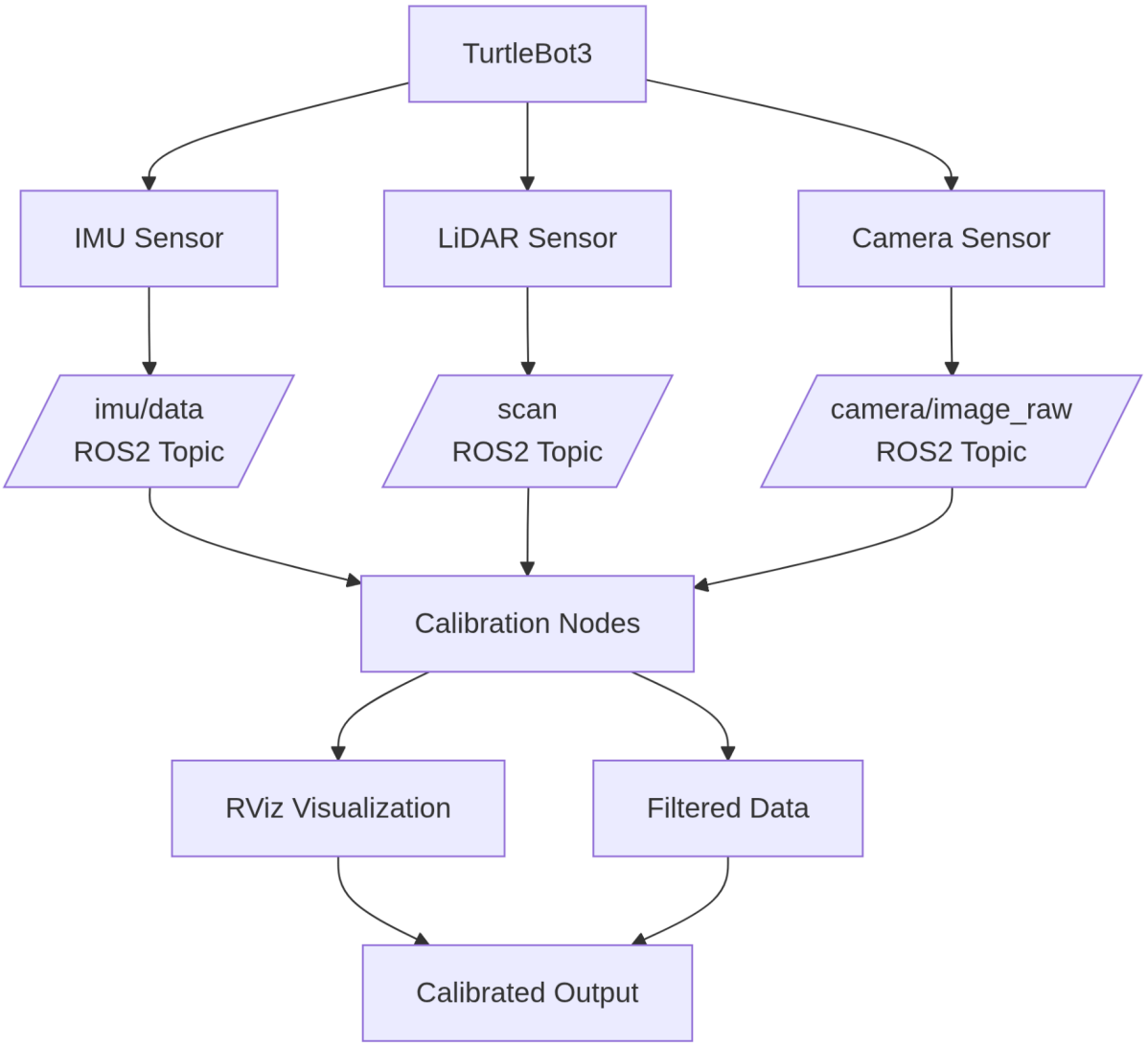
Sensor calibration is essential for accurate robot perception and navigation. Uncalibrated sensors can lead to drift, inaccurate measurements, and poor robot performance. This tutorial simplifies sensor calibration using TurtleBot3, which comes with pre-configured IMU, LiDAR, and optional camera sensors.

2. Learning Objectives

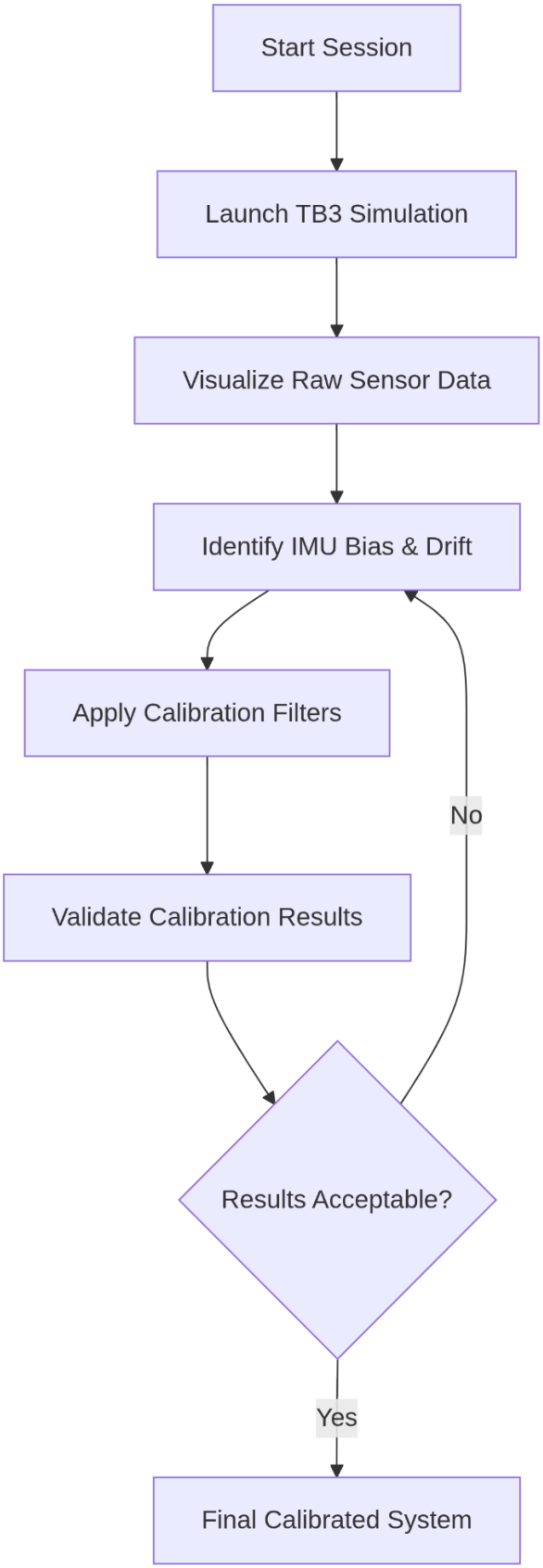
By the end of this training, you will be able to:

1. Understand the importance of sensor calibration in robotics
2. Visualize IMU, LiDAR, and camera data in RViz
3. Detect IMU bias and drift using ROS2 tools
4. Implement basic IMU calibration techniques
5. Process LiDAR data for obstacle detection
6. Display and analyze camera images
7. Use command-line tools for sensor monitoring
8. Create full example for calibration

3. Session Workflow Summary



4. IMU Calibration Section


Start Session
Launch TB3 Simulation
Visualize Raw Sensor Data
Identify IMU Bias & Drift
Apply Calibration Filters
Validate Calibration Results
Results Acceptable?
No
Yes
Final Calibrated System

What is IMU and Why Calibrate?

The Inertial Measurement Unit (IMU) measures:

- Angular velocity (gyroscope)
- Linear acceleration (accelerometer)
- Orientation (magnetometer - if available)

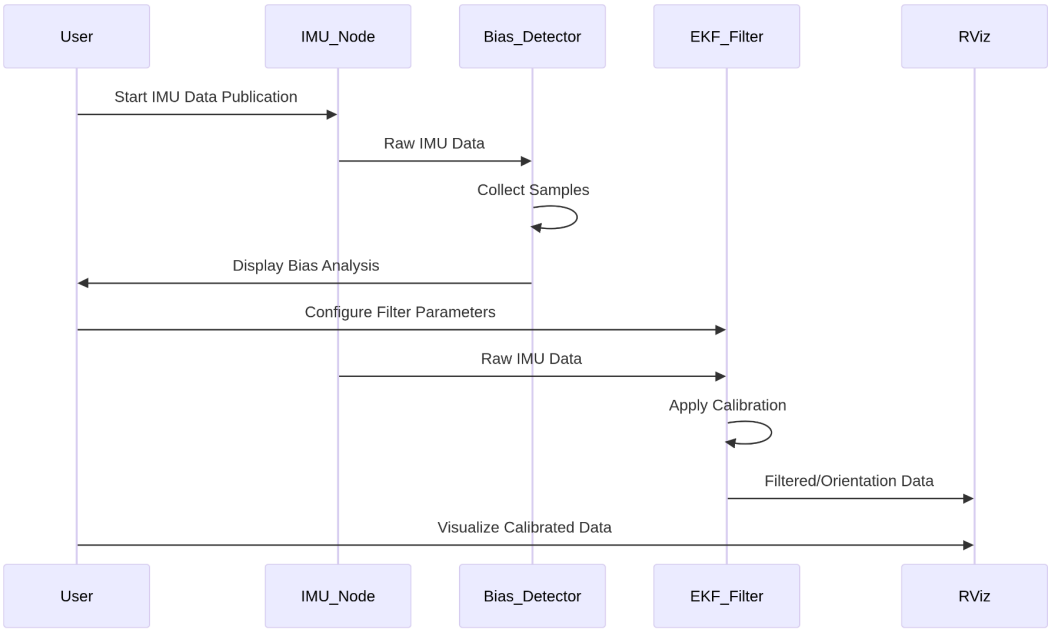
Common IMU Issues:

- Bias offset: Constant error in measurements
- Scale factor errors: Incorrect scaling of values
- Axis misalignment: Sensors not perfectly aligned
- Temperature drift: Values change with temperature

Checking IMU Data

```
bash
# Check IMU topic data
ros2 topic echo /imu/data

# Check publication rate
ros2 topic hz /imu/data

# View IMU in RViz
rviz2
# Add -> By topic -> /imu -> IMU
```

IMU Calibration Workflow



5. LiDAR Visualization & Basic Analysis

LaserScan Message Structure

The `/scan` topic publishes `sensor_msgs/LaserScan` messages containing:

- angle_min/max: Scan angle range
- angle_increment: Angle between measurements
- range_min/max: Valid distance range
- ranges: Array of distance measurements
- intensities: Reflection intensities (optional)

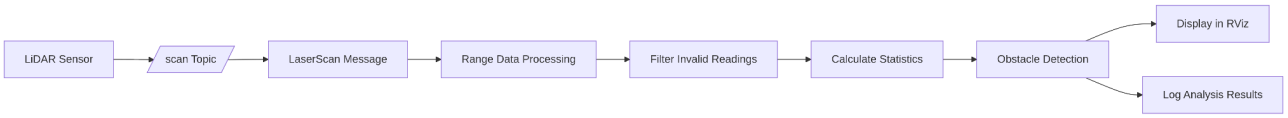
LiDAR Visualization in RViz

```
bash
```

```
# Launch RViz with LiDAR display
rviz2
```

```
# Add -> By topic -> /scan -> LaserScan
```

LiDAR Data Processing Flow



6. Camera Visualization Section (Optional)

Camera Data Display

```bash
# View camera feed using rqt_image_view
ros2 run rqt_image_view rqt_image_view

# Or view in RViz
rviz2
# Add -> By topic -> /camera/image_raw -> Camera
```

7. TurtleBot3 Command Line Tools

Essential Commands

```bash
# Set TurtleBot3 model
export TURTLEBOT3_MODEL=burger

# Launch simulation
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py

# View camera images
ros2 run rqt_image_view rqt_image_view

# Teleoperate the robot
ros2 run teleop_twist_keyboard teleop_twist_keyboard

# Monitor sensor topics
ros2 topic echo /scan
ros2 topic echo /imu/data
ros2 topic echo /camera/image_raw
```

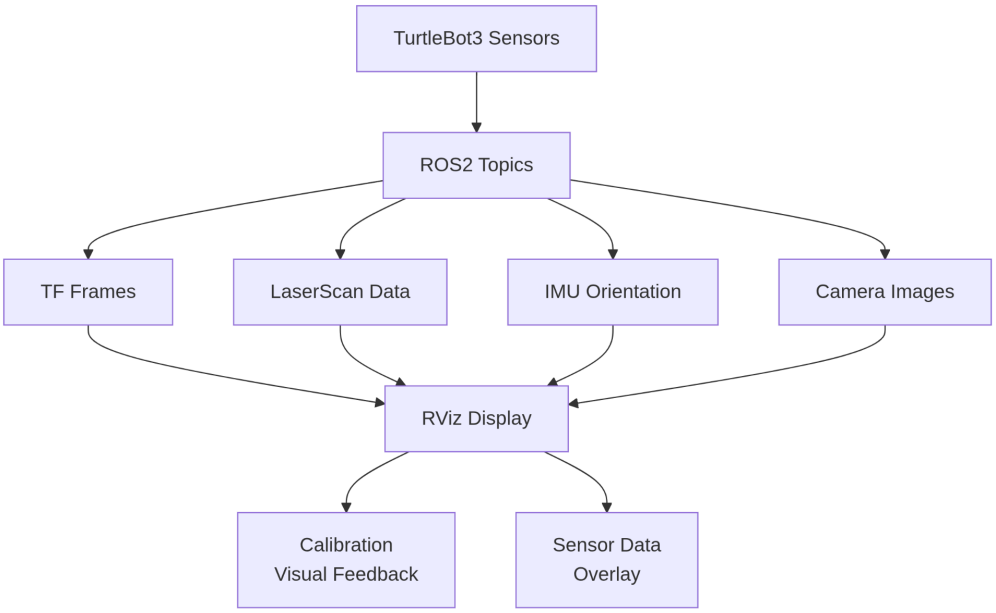
```
# Topic information
ros2 topic list
ros2 topic info /scan
ros2 topic hz /imu/data

# View TF tree
ros2 run tf2_tools view_frames.py
```

RViz Calibration Setup

RViz Visualization Configuration

1. Launch RViz:
2. `bash`
3. `rviz2`
4. Add Displays:
    - TF: Shows coordinate frames
    - RobotModel: Displays TurtleBot3 model
    - LaserScan: Visualize LiDAR data
    - IMU: Show orientation data
    - Camera: Display camera feed
5. Configure Fixed Frame:
    - Set to `odom` or `base_link`

RViz Visualization Pipeline



8.Create full example for calibration

Prerequisites

Required Software

- ROS2 Distribution: Humble or Iron
- TurtleBot3 Packages:
- bash

```
sudo apt install ros-${ROS_DISTRO}-turtlebot3*
```

- sudo apt install ros-${ROS_DISTRO}-gazebo-ros-pkgs
- Python Dependencies:
- bash
- pip install opencv-python numpy

Verify Installation

```
bash
```

```
# Check ROS2 installation

ros2 doctor



# Check TurtleBot3 packages

ros2 pkg list | grep turtlebot3



# Check Gazebo

gazebo --version
```

Package Setup

Step 1: Create Workspace and Package

```
bash

# Create workspace

mkdir -p ~/tb3_calibration_ws/src

cd ~/tb3_calibration_ws/src



# Create package

ros2 pkg create tb3_sensor_calibration \
```

```
  --build-type ament_python \
  --dependencies rclpy sensor_msgs geometry_msgs std_msgs cv_bridge
image_transport tf2_ros visualization_msgs
```

```
cd tb3_sensor_calibration
```

Step 2: Create Directory Structure

```
bash
```

```
# Create necessary directories
mkdir -p launch config scripts
mkdir -p tb3_sensor_calibration

# Create empty Python files
touch tb3_sensor_calibration/__init__.py
touch tb3_sensor_calibration/imu_bias_detector.py
touch tb3_sensor_calibration/imu_calibrator.py
touch tb3_sensor_calibration/lidar_analyzer.py
touch tb3_sensor_calibration/camera_viewer.py
```

```
touch tb3_sensor_calibration/sensor_integration.py



# Create configuration files

touch launch/calibration_launch.py

touch config/calibration.rviz

touch scripts/setup_calibration.sh

touch scripts/run_visualization.sh



# Make scripts executable

chmod +x scripts/*.sh
```

Step 3: Package Configuration Files

package.xml

```
xml

<?xml version="1.0"?>

<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypelayout="1.0"?>

<package format="3">

  <name>tb3_sensor_calibration</name>
```

```
  <version>1.0.0</version>

  <description>TurtleBot3 Sensor Calibration and Visualization
Package</description>

  <maintainer email="user@example.com">User</maintainer>

  <license>Apache License 2.0</license>



  <depend>rclpy</depend>

  <depend>sensor_msgs</depend>

  <depend>geometry_msgs</depend>

  <depend>std_msgs</depend>

  <depend>cv_bridge</depend>

  <depend>image_transport</depend>

  <depend>tf2_ros</depend>

  <depend>visualization_msgs</depend>

  <depend>turtlebot3_gazebo</depend>



  <exec_depend>ros2launch</exec_depend>
```

```
  <exec_depend>rviz2</exec_depend>

  <exec_depend>rqt_image_view</exec_depend>



  <export>

    <build_type>ament_python</build_type>

  </export>
</package>
```

setup.py

```
python
```

```
from setuptools import setup

import os

from glob import glob



package_name = 'tb3_sensor_calibration'



setup(

    name=package_name,
```

```
    version='1.0.0',

    packages=[package_name],

    data_files=[

        ('share/ament_index/resource_index/packages',

            ['resource/' + package_name]),

        ('share/' + package_name, ['package.xml']),

        (os.path.join('share', package_name, 'launch'),
glob('launch/*.py')),

        (os.path.join('share', package_name, 'config'),
glob('config/*.rviz')),

        (os.path.join('share', package_name, 'scripts'),
glob('scripts/*.sh')),

    ],

    install_requires=['setuptools'],

    zip_safe=True,

    maintainer='user',

    maintainer_email='user@example.com',

    description='TurtleBot3 Sensor Calibration and Visualization',
```

```
    license='Apache License 2.0',

    tests_require=['pytest'],

    entry_points={

        'console_scripts': [

            'imu_bias_detector =
tb3_sensor_calibration.imu_bias_detector:main',

            'imu_calibrator = tb3_sensor_calibration.imu_calibrator:main',

            'lidar_analyzer = tb3_sensor_calibration.lidar_analyzer:main',

            'camera_viewer = tb3_sensor_calibration.camera_viewer:main',

            'sensor_integration =
tb3_sensor_calibration.sensor_integration:main',

        ],

    },
)
```

Complete Node Implementations

1. IMU Bias Detector Node

File: `tb3_sensor_calibration/imu_bias_detector.py`

```
python
```

```python
#!/usr/bin/env python3

import rclpy

from rclpy.node import Node

from sensor_msgs.msg import Imu

import numpy as np

import json

import os

from rclpy.qos import QoSProfile, ReliabilityPolicy



class IMUBiasDetector(Node):

    def __init__(self):

        super().__init__('imu_bias_detector')



        # Declare parameters

        self.declare_parameter('imu_topic', '/imu')

        self.declare_parameter('max_samples', 500)
```

```
        self.declare_parameter('calibration_file', 'imu_calibration.json')

        # Get parameters
        imu_topic = self.get_parameter('imu_topic').value
        self.max_samples = self.get_parameter('max_samples').value
        self.calibration_file =
self.get_parameter('calibration_file').value

        # Statistics storage
        self.accel_data = []
        self.gyro_data = []
        self.sample_count = 0

        # Use best effort QoS for simulation
        qos_profile = QoSProfile(
            depth=10,
            reliability=ReliabilityPolicy.BEST_EFFORT
```

```
        )


        # Subscribe to IMU topic
        self.subscription = self.create_subscription(
            Imu,
            imu_topic,
            self.imu_callback,
            qos_profile)


        self.get_logger().info(f'🚀 IMU Bias Detector Started')
        self.get_logger().info(f'📡 Listening to: {imu_topic}')
        self.get_logger().info(f'📊 Collecting {self.max_samples}
samples...')
        self.get_logger().info('💡 Keep the robot STATIONARY during
calibration!')


    def imu_callback(self, msg):
```

```
if self.sample_count < self.max_samples:

    # Collect accelerometer data

    accel = [

        msg.linear_acceleration.x,

        msg.linear_acceleration.y,

        msg.linear_acceleration.z

    ]

    self.accel_data.append(accel)



    # Collect gyroscope data

    gyro = [

        msg.angular_velocity.x,

        msg.angular_velocity.y,

        msg.angular_velocity.z

    ]

    self.gyro_data.append(gyro)
```

```
            self.sample_count += 1

            # Progress updates
            if self.sample_count % 50 == 0:
                progress = (self.sample_count / self.max_samples) * 100
                self.get_logger().info(f'📈 Progress: {self.sample_count}/{self.max_samples} ({progress:.1f}%)')

        else:
            # Calculate and display bias
            self.calculate_bias()
            self.save_calibration()
            self.get_logger().info('🎉 IMU calibration completed!')
            self.get_logger().info('💾 Calibration data saved successfully')
            rclpy.shutdown()
```

```python
def calculate_bias(self):

    accel_array = np.array(self.accel_data)

    gyro_array = np.array(self.gyro_data)



    # Calculate statistics

    self.accel_bias = np.mean(accel_array, axis=0)

    self.accel_std = np.std(accel_array, axis=0)

    self.gyro_bias = np.mean(gyro_array, axis=0)

    self.gyro_std = np.std(gyro_array, axis=0)



    # Display results

    self.get_logger().info('=' * 50)

    self.get_logger().info('📋 IMU CALIBRATION RESULTS')

    self.get_logger().info('=' * 50)

    self.get_logger().info('🎯 Accelerometer Bias:')
```

```
self.get_logger().info(f'   X: {self.accel_bias[0]:.6f} m/s²')
self.get_logger().info(f'   Y: {self.accel_bias[1]:.6f} m/s²')
self.get_logger().info(f'   Z: {self.accel_bias[2]:.6f} m/s²')
self.get_logger().info('📊 Accelerometer Std Dev:')
self.get_logger().info(f'   X: {self.accel_std[0]:.6f} m/s²')
self.get_logger().info(f'   Y: {self.accel_std[1]:.6f} m/s²')
self.get_logger().info(f'   Z: {self.accel_std[2]:.6f} m/s²')
self.get_logger().info('🎯 Gyroscope Bias:')
self.get_logger().info(f'   X: {self.gyro_bias[0]:.6f} rad/s')
self.get_logger().info(f'   Y: {self.gyro_bias[1]:.6f} rad/s')
self.get_logger().info(f'   Z: {self.gyro_bias[2]:.6f} rad/s')
self.get_logger().info('📊 Gyroscope Std Dev:')
self.get_logger().info(f'   X: {self.gyro_std[0]:.6f} rad/s')
self.get_logger().info(f'   Y: {self.gyro_std[1]:.6f} rad/s')
self.get_logger().info(f'   Z: {self.gyro_std[2]:.6f} rad/s')
```

```
        # Interpretation
        self.get_logger().info('=' * 50)
        self.get_logger().info('💡 INTERPRETATION:')
        if abs(self.accel_bias[2] - 9.8) < 0.5:
            self.get_logger().info('✅ Z-axis accelerometer shows expected gravity value')
        else:
            self.get_logger().warning('⚠️  Z-axis accelerometer may need manual calibration')


    def save_calibration(self):
        calibration_data = {
            'accel_bias': self.accel_bias.tolist(),
            'accel_std': self.accel_std.tolist(),
            'gyro_bias': self.gyro_bias.tolist(),
            'gyro_std': self.gyro_std.tolist(),
            'samples_collected': self.sample_count,
```

```
            'timestamp': self.get_clock().now().to_msg().sec
        }

        with open(self.calibration_file, 'w') as f:
            json.dump(calibration_data, f, indent=4)

        self.get_logger().info(f'💾 Data saved to:
{os.path.abspath(self.calibration_file)}')

def main():
    rclpy.init()
    node = IMUBiasDetector()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.get_logger().info('❌ Calibration interrupted by user')
    finally:
```

```
        node.destroy_node()

        rclpy.shutdown()



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

2. IMU Calibrator Node

File: `tb3_sensor_calibration/imu_calibrator.py`

```
python

#!/usr/bin/env python3

import rclpy

from rclpy.node import Node

from sensor_msgs.msg import Imu

import numpy as np

import json

import os

from rclpy.qos import QoSProfile, ReliabilityPolicy
```

```
class IMUCalibrator(Node):

    def __init__(self):

        super().__init__('imu_calibrator')



        # Declare parameters

        self.declare_parameter('imu_topic', '/imu')

        self.declare_parameter('calibration_file', 'imu_calibration.json')

        self.declare_parameter('output_topic', '/imu/calibrated')



        # Get parameters

        imu_topic = self.get_parameter('imu_topic').value

        self.calibration_file =
self.get_parameter('calibration_file').value

        output_topic = self.get_parameter('output_topic').value



        # Load calibration data

        self.calibration_data = self.load_calibration()
```

```
# Use best effort QoS for simulation
qos_profile = QoSProfile(
    depth=10,
    reliability=ReliabilityPolicy.BEST_EFFORT
)

# Publishers and subscribers
self.publisher = self.create_publisher(Imu, output_topic, 10)
self.subscription = self.create_subscription(
    Imu,
    imu_topic,
    self.imu_callback,
    qos_profile)

if self.calibration_data:
```

```
            self.get_logger().info('🚀 IMU Calibrator Started')
            self.get_logger().info(f'📡 Input: {imu_topic}')
            self.get_logger().info(f'📤 Output: {output_topic}')
            self.get_logger().info('✅ Calibration data loaded
successfully')
        else:
            self.get_logger().error('❌ Failed to load calibration data!')


    def load_calibration(self):
        try:
            with open(self.calibration_file, 'r') as f:
                data = json.load(f)
                self.get_logger().info(f'📂 Loaded calibration from:
{self.calibration_file}')
                return data
        except FileNotFoundError:
            self.get_logger().error(f'❌ Calibration file not found:
{self.calibration_file}')
```

```
            self.get_logger().info('💡 Run imu_bias_detector first to
generate calibration data')

            return None

        except json.JSONDecodeError:

            self.get_logger().error(f'❌ Invalid calibration file:
{self.calibration_file}')

            return None



    def imu_callback(self, msg):

        if not self.calibration_data:

            return



        # Create calibrated message

        calibrated_msg = Imu()

        calibrated_msg.header = msg.header

        calibrated_msg.header.frame_id = 'imu_link_calibrated'
```

```
        # Apply gyroscope calibration

        calibrated_msg.angular_velocity.x = msg.angular_velocity.x -
self.calibration_data['gyro_bias'][0]

        calibrated_msg.angular_velocity.y = msg.angular_velocity.y -
self.calibration_data['gyro_bias'][1]

        calibrated_msg.angular_velocity.z = msg.angular_velocity.z -
self.calibration_data['gyro_bias'][2]



        # Apply accelerometer calibration

        calibrated_msg.linear_acceleration.x = msg.linear_acceleration.x -
self.calibration_data['accel_bias'][0]

        calibrated_msg.linear_acceleration.y = msg.linear_acceleration.y -
self.calibration_data['accel_bias'][1]

        calibrated_msg.linear_acceleration.z = msg.linear_acceleration.z -
self.calibration_data['accel_bias'][2]



        # Copy other fields

        calibrated_msg.orientation = msg.orientation

        calibrated_msg.orientation_covariance = msg.orientation_covariance
```

```
        calibrated_msg.angular_velocity_covariance =
msg.angular_velocity_covariance

        calibrated_msg.linear_acceleration_covariance =
msg.linear_acceleration_covariance



        # Publish calibrated data

        self.publisher.publish(calibrated_msg)



def main():

    rclpy.init()

    node = IMUCalibrator()

    try:

        rclpy.spin(node)

    except KeyboardInterrupt:

        node.get_logger().info('❌ IMU calibrator stopped by user')

    finally:

        node.destroy_node()
```

```
        rclpy.shutdown()


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

3. LiDAR Analyzer Node

File: `tb3_sensor_calibration/lidar_analyzer.py`

```
python

#!/usr/bin/env python3

import rclpy

from rclpy.node import Node

from sensor_msgs.msg import LaserScan

import numpy as np

from visualization_msgs.msg import Marker, MarkerArray

from geometry_msgs.msg import Point

from std_msgs.msg import ColorRGBA

from rclpy.qos import QoSProfile, ReliabilityPolicy
```

```
class LidarAnalyzer(Node):

    def __init__(self):

        super().__init__('lidar_analyzer')



        # Use best effort QoS for simulation

        qos_profile = QoSProfile(

            depth=10,

            reliability=ReliabilityPolicy.BEST_EFFORT

        )



        self.subscription = self.create_subscription(

            LaserScan,

            '/scan',

            self.lidar_callback,

            qos_profile)
```

```
        # Publisher for visualization markers
        self.marker_pub = self.create_publisher(MarkerArray,
'/lidar_markers', 10)

        self.get_logger().info('🚀 LiDAR Analyzer Started')
        self.get_logger().info('📡 Listening to /scan topic')

    def lidar_callback(self, msg):
        # Convert ranges to numpy array for processing
        ranges = np.array(msg.ranges)

        # Filter out invalid measurements
        valid_ranges = ranges[(ranges > msg.range_min) & (ranges <
msg.range_max)]

        if len(valid_ranges) > 0:
            min_distance = np.min(valid_ranges)
```

```
            max_distance = np.max(valid_ranges)

            avg_distance = np.mean(valid_ranges)



            # Find angle and index of closest object

            min_index = np.argmin(ranges)

            angle_to_closest = msg.angle_min + (min_index *
msg.angle_increment)



            # Calculate position of closest object

            closest_x = min_distance * np.cos(angle_to_closest)

            closest_y = min_distance * np.sin(angle_to_closest)



            # Log statistics

            self.get_logger().info(

                f'📊 LiDAR - Min: {min_distance:.2f}m | '

                f'Max: {max_distance:.2f}m | '

                f'Avg: {avg_distance:.2f}m | '
```

```
                f'Closest: ({closest_x:.2f}, {closest_y:.2f})'

            )



            # Publish visualization markers

            self.publish_markers(closest_x, closest_y, min_distance)



            # Obstacle detection warnings

            if min_distance < 0.3:  # 30cm threshold

                self.get_logger().warning(f'🚨 CLOSE OBSTACLE!
{min_distance:.2f}m')

            elif min_distance < 0.5:  # 50cm threshold

                self.get_logger().warning(f'⚠️  Obstacle at
{min_distance:.2f}m')



        else:

            self.get_logger().warning('❌ No valid LiDAR measurements!')
```

```
def publish_markers(self, x, y, distance):

    marker_array = MarkerArray()



    # Marker for closest point

    marker = Marker()

    marker.header.frame_id = "base_scan"

    marker.header.stamp = self.get_clock().now().to_msg()

    marker.ns = "closest_point"

    marker.id = 0

    marker.type = Marker.SPHERE

    marker.action = Marker.ADD



    marker.pose.position.x = x

    marker.pose.position.y = y

    marker.pose.position.z = 0.0

    marker.pose.orientation.w = 1.0
```

```
marker.scale.x = 0.1

marker.scale.y = 0.1

marker.scale.z = 0.1



# Color based on distance (red = close, green = far)

marker.color.a = 1.0

if distance < 0.5:

    marker.color.r = 1.0

    marker.color.g = 0.0

    marker.color.b = 0.0

else:

    marker.color.r = 0.0

    marker.color.g = 1.0

    marker.color.b = 0.0
```

```
        marker_array.markers.append(marker)

        self.marker_pub.publish(marker_array)



def main():

    rclpy.init()

    node = LidarAnalyzer()

    try:

        rclpy.spin(node)

    except KeyboardInterrupt:

        node.get_logger().info('❌ LiDAR analyzer stopped by user')

    finally:

        node.destroy_node()

        rclpy.shutdown()



if __name__ == '__main__':

    main()
```

4. Camera Viewer Node

File: `tb3_sensor_calibration/camera_viewer.py`

```
python
```

```python
#!/usr/bin/env python3

import rclpy

from rclpy.node import Node

from sensor_msgs.msg import Image

from cv_bridge import CvBridge

import cv2

import numpy as np

from rclpy.qos import QoSProfile, ReliabilityPolicy



class CameraViewer(Node):

    def __init__(self):

        super().__init__('camera_viewer')



        self.bridge = CvBridge()
```

```
        # Use best effort QoS for simulation
        qos_profile = QoSProfile(
            depth=10,
            reliability=ReliabilityPolicy.BEST_EFFORT
        )

        self.subscription = self.create_subscription(
            Image,
            '/camera/image_raw',
            self.image_callback,
            qos_profile)

        # Publisher for processed image
        self.processed_pub = self.create_publisher(Image,
'/camera/processed', 10)
```

```
        self.get_logger().info('🚀 Camera Viewer Started')
        self.get_logger().info('📡 Listening to /camera/image_raw')
        self.get_logger().info('💡 Press Q to close camera windows')

    def image_callback(self, msg):
        try:
            # Convert ROS Image message to OpenCV image
            cv_image = self.bridge.imgmsg_to_cv2(msg, 'bgr8')

            # Get image dimensions
            height, width, channels = cv_image.shape

            # Display basic info (only once)
            if not hasattr(self, 'info_displayed'):
                self.get_logger().info(f'📷 Camera resolution:
{width}x{height}')
                self.info_displayed = True
```

```
            # Image processing pipeline
            processed_image = self.process_image(cv_image)

            # Display both images
            cv2.imshow('Raw Camera Feed', cv_image)
            cv2.imshow('Processed Image (Edge Detection)', processed_image)

            # Convert back to ROS message and publish
            processed_msg = self.bridge.cv2_to_imgmsg(processed_image,
'bgr8')
            processed_msg.header = msg.header
            self.processed_pub.publish(processed_msg)

            # Check for 'q' key to quit
            key = cv2.waitKey(1) & 0xFF
            if key == ord('q'):
```

```
                self.get_logger().info('❌ Closing camera viewer...')

                cv2.destroyAllWindows()

                raise KeyboardInterrupt



        except Exception as e:

            self.get_logger().error(f'❌ Error processing image: {str(e)}')



def process_image(self, image):

    # Convert to grayscale

    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)



    # Edge detection

    edges = cv2.Canny(gray, 50, 150)



    # Convert edges to color for display

    edges_colored = cv2.cvtColor(edges, cv2.COLOR_GRAY2BGR)
```

```
        # Add text info
        cv2.putText(edges_colored, 'Edge Detection', (10, 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
        cv2.putText(edges_colored, 'Press Q to quit', (10, 70),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)

        return edges_colored


def main():
    rclpy.init()
    node = CameraViewer()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
```

```
    finally:
        cv2.destroyAllWindows()
        node.destroy_node()
        rclpy.shutdown()


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

5. Main Launch File

File: `launch/calibration_launch.py`

```
python
#!/usr/bin/env python3

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

```
def generate_launch_description():


    # Set TurtleBot3 model

    set_tb3_model = SetEnvironmentVariable('TURTLEBOT3_MODEL', 'burger')



    # Get package share directories

    tb3_sensor_calibration_dir =
get_package_share_directory('tb3_sensor_calibration')

    turtlebot3_gazebo_dir =
get_package_share_directory('turtlebot3_gazebo')



    return LaunchDescription([

        set_tb3_model,



        # Launch TurtleBot3 simulation

        IncludeLaunchDescription(
```

```
            PythonLaunchDescriptionSource(
                os.path.join(turtlebot3_gazebo_dir, 'launch',
'turtlebot3_world.launch.py')
            )
        ),


        # IMU Bias Detector Node
        Node(
            package='tb3_sensor_calibration',
            executable='imu_bias_detector',
            name='imu_bias_detector',
            output='screen',
            parameters=[{
                'imu_topic': '/imu',
                'max_samples': 500
            }]
        ),
```

```
# IMU Calibrator Node
Node(
    package='tb3_sensor_calibration',
    executable='imu_calibrator',
    name='imu_calibrator',
    output='screen',
    parameters=[{
        'imu_topic': '/imu',
        'output_topic': '/imu/calibrated'
    }]
),

# LiDAR Analyzer Node
Node(
    package='tb3_sensor_calibration',
```

```
        executable='lidar_analyzer',
        name='lidar_analyzer',
        output='screen'
    ),

    # Camera Viewer Node
    Node(
        package='tb3_sensor_calibration',
        executable='camera_viewer',
        name='camera_viewer',
        output='screen'
    ),

    # RViz2 with configuration
    Node(
        package='rviz2',
```

```
            executable='rviz2',

            name='rviz2',

            arguments=['-d', os.path.join(tb3_sensor_calibration_dir,
'config', 'calibration.rviz')],

            output='screen'

        )
    ])
```

Quick Start

Method 1: Complete System (Recommended)

```
bash

# Terminal 1 - Build and run complete system

cd ~/tb3_calibration_ws

colcon build --packages-select tb3_sensor_calibration

source install/setup.bash

export TURTLEBOT3_MODEL=burger

ros2 launch tb3_sensor_calibration calibration_launch.py
```

Method 2: Step-by-Step Manual Approach

```
bash
```

```
# Terminal 1 - Start simulation only

source ~/tb3_calibration_ws/install/setup.bash

export TURTLEBOT3_MODEL=burger

ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py



# Terminal 2 - IMU calibration

source ~/tb3_calibration_ws/install/setup.bash

ros2 run tb3_sensor_calibration imu_bias_detector --ros-args -p
imu_topic:=/imu



# Terminal 3 - IMU calibrator (after bias detection completes)

source ~/tb3_calibration_ws/install/setup.bash

ros2 run tb3_sensor_calibration imu_calibrator --ros-args -p
imu_topic:=/imu



# Terminal 4 - LiDAR analysis

source ~/tb3_calibration_ws/install/setup.bash
```

```
ros2 run tb3_sensor_calibration lidar_analyzer
```

```
# Terminal 5 - Camera viewer
source ~/tb3_calibration_ws/install/setup.bash
ros2 run tb3_sensor_calibration camera_viewer
```

```
# Terminal 6 - RViz
source ~/tb3_calibration_ws/install/setup.bash
rviz2
```

Detailed Calibration Steps

Step 1: IMU Calibration Process

Step 2: Expected IMU Calibration Results

Normal IMU Values (Stationary Robot):

- Accelerometer Z-axis: ~9.8 m/s² (gravity)
- Accelerometer X,Y-axis: ~0.0 m/s²
- Gyroscope all axes: ~0.0 rad/s

After Calibration:

- Accelerometer Z-axis: ~0.0 m/s² (gravity compensated)

- All biases subtracted from raw measurements

Step 3: Verification Commands

```bash
# Check calibration file

cat ~/tb3_calibration_ws/imu_calibration.json



# Compare raw vs calibrated IMU

ros2 topic echo /imu --field linear_acceleration | head -n 5

ros2 topic echo /imu/calibrated --field linear_acceleration | head -n 5



# Monitor LiDAR

ros2 topic echo /lidar_markers



# Check camera topics

ros2 topic hz /camera/image_raw
```

Troubleshooting Guide

Common Build Issues

Issue: "Package not found" after building

```
bash

# Solution: Source the workspace properly

source ~/tb3_calibration_ws/install/setup.bash



# Verify package exists

ros2 pkg list | grep tb3_sensor_calibration
```

Issue: "Executable not found"

```
bash

# Solution: Rebuild and check setup.py

cd ~/tb3_calibration_ws

colcon build --packages-select tb3_sensor_calibration



# Check entry points in setup.py

ros2 pkg executables tb3_sensor_calibration
```

Common Runtime Issues

Issue: No IMU data received

```bash
# Check available topics

ros2 topic list | grep imu



# Common topic names: /imu, /imu/data, /imu_raw

# Adjust parameter accordingly:

ros2 run tb3_sensor_calibration imu_bias_detector --ros-args -p
imu_topic:=/imu
```

Issue: Gazebo not starting

```bash
# Check Gazebo installation

gazebo --version



# Set model explicitly

export TURTLEBOT3_MODEL=burger



# Launch with verbose output

ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py --verbose
```

Issue: Camera windows not opening

```bash
# Check if OpenCV can display windows

# If using SSH or headless, use image transport instead:
ros2 run rqt_image_view rqt_image_view
```

QoS Configuration Issues

Issue: "No data received" in simulation

- Cause: Gazebo uses BEST_EFFORT reliability by default
- Solution: Our nodes now use BEST_EFFORT QoS profile

Common Issues & Solutions

1. IMU Calibration Not Starting

Symptoms: No progress messages, stuck at "Collecting samples..."
Solutions:

```bash
# Check if IMU topic has data
ros2 topic echo /imu --field angular_velocity | head -n 5

# Check topic rate
ros2 topic hz /imu
```

```
# Try different topic names

ros2 topic list | grep imu
```

2. LiDAR No Data

Symptoms: "No valid LiDAR measurements" warning

Solutions:

```
bash

# Check if robot is in the world with obstacles

ros2 topic echo /scan --field ranges | head -n 5



# Check LiDAR topic

ros2 topic info /scan
```

3. Camera Issues

Symptoms: No camera windows or errors

Solutions:

```
bash

# Check camera topic

ros2 topic hz /camera/image_raw
```

```
# Use rqt_image_view as alternative

ros2 run rqt_image_view rqt_image_view
```

```
# Check if OpenCV is installed

python3 -c "import cv2; print(cv2.__version__)"
```

4. RViz Configuration

Symptoms: No data in RViz or TF errors
Solutions:

- Set fixed frame to "base_link" or "odom"
- Check TF tree: `ros2 run tf2_tools view_frames.py`
- **Add displays manually for each topic**

Advanced Configuration

Customizing Calibration Parameters

Modify sample count for faster/slower calibration:

```
bash

ros2 run tb3_sensor_calibration imu_bias_detector --ros-args -p
max_samples:=200
```

Change calibration file location:

```
bash

ros2 run tb3_sensor_calibration imu_bias_detector --ros-args -p
calibration_file:=/path/to/calibration.json
```

Using with Real TurtleBot3

For real robot usage, modify the launch file:

```
python
```

```
# Instead of Gazebo launch, use:

Node(
    package='turtlebot3_bringup',
    executable='robot_launch.py',
    name='robot_launch'
)
```

Validation & Testing

Test 1: IMU Calibration Validation

```
bash
```

```
# Before calibration: Z-axis should show ~9.8

ros2 topic echo /imu --field linear_acceleration.z



# After calibration: Z-axis should show ~0.0 when stationary

ros2 topic echo /imu/calibrated --field linear_acceleration.z
```

Test 2: LiDAR Functionality

```
bash

# Move robot and check distance changes

ros2 run teleop_twist_keyboard teleop_twist_keyboard




# Monitor LiDAR warnings for obstacles

ros2 topic echo /lidar_markers
```

Test 3: Camera Processing

```
bash

# Verify edge detection is working

ros2 run rqt_image_view rqt_image_view

# Subscribe to /camera/processed
```

Expected Performance Metrics

- IMU Calibration: Completes in 10-30 seconds (500 samples)
- LiDAR Update Rate: 5-10 Hz
- Camera Processing: 15-30 FPS depending on system
- Overall System: Stable with all nodes running simultaneously

Conclusion

This simplified training version demonstrates the essential concepts of ROS2 sensor calibration and visualization using TurtleBot3. By focusing on practical implementation rather than complex URDF creation, beginners can quickly understand and apply sensor calibration techniques.

Key Takeaways:

- Sensor calibration is crucial for accurate robot operation
- ROS2 provides excellent tools for sensor data visualization and analysis
- TurtleBot3 offers a complete platform for learning sensor integration
- Regular calibration improves long-term robot performance

Continue practicing with different sensor configurations and explore advanced filtering techniques like Kalman filters for improved sensor fusion.