ROS2 Sensor Integration & Calibration

Section 1: Sensor Fundamentals & Data Visualization

Introduction

In this session, we'll explore the fundamental principles of robot sensors and learn how to integrate them into a ROS2 system. Sensors are the "eyes and ears" of your robot, providing crucial data about the environment. Understanding how to properly set up, visualize, and interpret sensor data is the first step toward building intelligent robotic systems.

Learning Objectives

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

- Understand the working principles of IMU, LiDAR, and Camera sensors
- Set up sensor plugins in Gazebo simulation
- Visualize sensor data in real-time using ROS2 tools
- Interpret sensor readings and understand their practical implications
- Identify common sensor issues and their symptoms

1.1 IMU (Inertial Measurement Unit) Fundamentals

What is an IMU?

An IMU is a combination of sensors that measures the robot's motion and orientation:

- Accelerometer: Measures linear acceleration in 3 axes (X, Y, Z)
- Gyroscope: Measures angular velocity (rotation rates) around 3 axes
- Magnetometer (optional): Measures magnetic field for absolute heading

Key Concepts:

- Gravity Vector: When stationary, accelerometer should read [0, 0, 9.81] m/s² (gravity)
- Stationary Condition: Gyroscope should read [0, 0, 0] rad/s when not rotating

- Orientation: Derived from integrating angular velocity over time
- Sensor Fusion: Combining multiple sensors for better accuracy

Expected Data Patterns:

- Stationary Robot:
  - Acceleration: ~[0, 0, 9.81] m/s²
  - Gyroscope: ~[0, 0, 0] rad/s
- Moving Forward:
  - Acceleration: Positive X-axis during acceleration
  - Gyroscope: Minimal rotation
- Turning:
  - Acceleration: Centripetal acceleration
  - Gyroscope: Significant Z-axis rotation

IMU Basics & Data Visualization

Objective

Set up a robot with IMU sensor in Gazebo, visualize IMU data in terminal, and understand why calibration is needed by observing sensor biases.

Project Structure

```
demo_sensor/
├── package.xml
├── setup.py
├── setup.cfg
├── launch/
│   └── simulation.launch.py
├── urdf/
│   └── simple_robot.urdf.xacro
└── demo_sensor/
    ├── __init__.py
    ├── imu_display.py
    └── robot_mover.py
```

File 1: Package Configuration

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>demo_sensor</name>
  <version>0.1.0</version>
  <description>IMU Sensor Demonstration - Shows IMU data and calibration need</description>
  <maintainer email="user@example.com">User</maintainer>
  <license>Apache-2.0</license>


  <depend>rclpy</depend>

  <depend>gazebo_ros</depend>

  <depend>geometry_msgs</depend>

  <depend>sensor_msgs</depend>

  <depend>robot_state_publisher</depend>


  <export>

    <build_type>ament_python</build_type>

  </export>

</package>
```

Purpose: Defines package dependencies and metadata.

setup.py

```python
from setuptools import setup


package_name = 'demo_sensor'


setup(

    name=package_name,

    version='0.1.0',

    packages=[package_name],

    data_files=[

        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        ('share/' + package_name + '/launch',
['launch/simulation.launch.py']),
        ('share/' + package_name + '/urdf',
['urdf/simple_robot.urdf.xacro']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='User',
    maintainer_email='user@example.com',
    description='IMU Sensor Demonstration',
    license='Apache-2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'imu_display = demo_sensor.imu_display:main',
            'robot_mover = demo_sensor.robot_mover:main',
        ],
    },
)
```

Purpose: Python package setup with executable nodes.

File 2: Robot URDF with IMU

urdf/simple_robot.urdf.xacro

```
xml
```

```xml
<?xml version="1.0"?>
<robot name="simple_robot" xmlns:xacro="http://www.ros.org/wiki/xacro">

  <!-- Base Link - Main body of the robot -->
  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.4 0.2 0.1"/>
      </geometry>
      <material name="blue">
        <color rgba="0 0 0.8 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <box size="0.4 0.2 0.1"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="2.0"/>
      <inertia ixx="0.1" ixy="0.0" ixz="0.0" iyy="0.1" iyz="0.0" izz="0.1"/>
    </inertial>
  </link>

  <!-- IMU Link - Red box representing IMU sensor -->
```

```xml
<link name="imu_link">
  <visual>
    <geometry>
      <box size="0.05 0.05 0.02"/>
    </geometry>
    <material name="red">
      <color rgba="1 0 0 1"/>
    </material>
  </visual>
</link>

<!-- IMU Joint - Fixed connection to base -->
<joint name="imu_joint" type="fixed">
  <parent link="base_link"/>
  <child link="imu_link"/>
  <origin xyz="0.1 0.0 0.05" rpy="0 0 0"/>
</joint>

<!-- Left Wheel -->
<link name="left_wheel">
  <visual>
    <geometry>
      <cylinder length="0.05" radius="0.06"/>
    </geometry>
    <material name="black">
      <color rgba="0 0 0 1"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.05" radius="0.06"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="0.5"/>
    <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0" izz="0.001"/>
  </inertial>
</link>

<!-- Right Wheel -->
<link name="right_wheel">
  <visual>
    <geometry>
      <cylinder length="0.05" radius="0.06"/>
    </geometry>
    <material name="black">
      <color rgba="0 0 0 1"/>
    </material>
  </visual>
  <collision>
    <geometry>
```

```xml
        <cylinder length="0.05" radius="0.06"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.5"/>
      <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0" izz="0.001"/>
    </inertial>
  </link>

  <!-- Caster Wheel -->
  <link name="caster_wheel">
    <visual>
      <geometry>
        <sphere radius="0.03"/>
      </geometry>
      <material name="gray">
        <color rgba="0.5 0.5 0.5 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <sphere radius="0.03"/>
      </geometry>
    </collision>
    <inertial>
```

```
      <mass value="0.1"/>
      <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0" izz="0.0001"/>
    </inertial>
  </link>

  <!-- Wheel Joints -->
  <joint name="left_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="left_wheel"/>
    <origin xyz="0.0 0.1 -0.025" rpy="1.5708 0 0"/>
    <axis xyz="0 0 1"/>
  </joint>

  <joint name="right_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="right_wheel"/>
    <origin xyz="0.0 -0.1 -0.025" rpy="1.5708 0 0"/>
    <axis xyz="0 0 1"/>
  </joint>

  <joint name="caster_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="caster_wheel"/>
    <origin xyz="-0.15 0.0 -0.05" rpy="0 0 0"/>
    <axis xyz="0 0 1"/>
  </joint>


  <!-- Gazebo Plugins -->

  <gazebo>

    <!-- Differential Drive Controller - Enables robot movement -->

    <plugin name="diff_drive_controller"
filename="libgazebo_ros_diff_drive.so">

      <command_topic>/cmd_vel</command_topic>

      <odometry_topic>/odom</odometry_topic>

      <odometry_frame>odom</odometry_frame>

      <robot_base_frame>base_link</robot_base_frame>

      <left_joint>left_wheel_joint</left_joint>

      <right_joint>right_wheel_joint</right_joint>

      <wheel_separation>0.2</wheel_separation>

      <wheel_diameter>0.12</wheel_diameter>

      <publish_odom>true</publish_odom>

      <publish_odom_tf>true</publish_odom_tf>

      <publish_wheel_tf>true</publish_wheel_tf>

      <max_wheel_torque>20</max_wheel_torque>

      <wheel_acceleration>1.0</wheel_acceleration>

    </plugin>

  </gazebo>


  <!-- IMU Sensor Plugin - Publishes IMU data to /imu/data topic -->

  <gazebo reference="imu_link">

    <sensor name="imu_sensor" type="imu">
```

```xml
    <always_on>true</always_on>
    <update_rate>100</update_rate>
    <imu>
      <topic>/imu/data</topic>
      <noise>
        <type>gaussian</type>
        <rate>
          <mean>0.0</mean>
          <stddev>0.0002</stddev>
          <bias_mean>0.0</bias_mean>
          <bias_stddev>0.00003</bias_stddev>
        </rate>
        <accel>
          <mean>0.0</mean>
          <stddev>0.017</stddev>
          <bias_mean>0.1</bias_mean>
          <bias_stddev>0.001</bias_stddev>
        </accel>
      </noise>
    </imu>
  </sensor>
</gazebo>

<!-- Gazebo Material Colors -->
<gazebo reference="base_link">
  <material>Gazebo/Blue</material>
  </gazebo>
  <gazebo reference="imu_link">
    <material>Gazebo/Red</material>
  </gazebo>
  <gazebo reference="left_wheel">
    <material>Gazebo/Black</material>
  </gazebo>
  <gazebo reference="right_wheel">
    <material>Gazebo/Black</material>
  </gazebo>
  <gazebo reference="caster_wheel">
    <material>Gazebo/Gray</material>
  </gazebo>

</robot>
```

Purpose: Defines robot physical structure with IMU sensor and Gazebo plugins.

File 3: IMU Data Display Node

demo_sensor/imu_display.py

```
python
```

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

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Imu
```

```python
import math


class IMUDisplay(Node):
    def __init__(self):
        super().__init__('imu_display')

        # Subscribe to IMU data topic
        self.subscription = self.create_subscription(
            Imu,
            '/imu/data',
            self.imu_callback,
            10)

        self.get_logger().info('=== IMU DATA DISPLAY STARTED ===')
        self.get_logger().info('Waiting for IMU data...')
        self.get_logger().info('When robot is STATIONARY, expect:')
        self.get_logger().info('  Acceleration: [0.0, 0.0, ~9.8] m/s²')
        self.get_logger().info('  Gyroscope:    [0.0, 0.0, 0.0] rad/s')
        self.get_logger().info('Any non-zero gyro or wrong accel values indicate CALIBRATION NEEDED!')

        self.sample_count = 0

    def imu_callback(self, msg):
        # Show first sample immediately
        if self.sample_count == 0:
            self.get_logger().info('✓ IMU DATA RECEIVED! Showing real-time data:')

        self.sample_count += 1

        # Only print every 50 samples to avoid spam
        if self.sample_count % 50 == 0:
            # Extract data from IMU message
            accel = msg.linear_acceleration
            gyro = msg.angular_velocity

            # Calculate orientation from quaternion
            orient = msg.orientation
            roll, pitch, yaw = self.quaternion_to_euler(orient.x, orient.y, orient.z, orient.w)

            print("\n" + "="*60)
            print("IMU SENSOR DATA - Sample #{}".format(self.sample_count))
            print("="*60)

            # Acceleration data with calibration check
            print("ACCELERATION (m/s²):")
            print("  X: {:8.4f} | Y: {:8.4f} | Z: {:8.4f}".format(accel.x, accel.y, accel.z))
            if abs(accel.z - 9.8) > 0.5:
                print("  ⚠️  Z should be ~9.8 (gravity) - CALIBRATION NEEDED!")
            if abs(accel.x) > 0.1 or abs(accel.y) > 0.1:
                print("  ⚠️  X/Y should be ~0.0 when stationary - CALIBRATION NEEDED!")

            # Gyroscope data with calibration check
            print("\nGYROSCOPE (rad/s):")
            print("  X: {:8.4f} | Y: {:8.4f} | Z: {:8.4f}".format(gyro.x, gyro.y, gyro.z))
            if abs(gyro.x) > 0.01 or abs(gyro.y) > 0.01 or abs(gyro.z) > 0.01:
                print("  ⚠️  Should be [0,0,0] when stationary - CALIBRATION NEEDED!")

            # Orientation data
            print("\nORIENTATION (degrees):")
            print("  Roll: {:6.1f}° | Pitch: {:6.1f}° | Yaw: {:6.1f}°".format(roll, pitch, yaw))

            # Overall calibration status
            print("\nCALIBRATION STATUS:")
            needs_calibration = False
            if abs(accel.z - 9.8) > 0.5:
                print("  ❌ Accelerometer needs calibration (gravity wrong)")
                needs_calibration = True
            if abs(accel.x) > 0.1 or abs(accel.y) > 0.1:
                print("  ❌ Accelerometer needs calibration (X/Y bias)")
                needs_calibration = True
            if abs(gyro.x) > 0.01 or abs(gyro.y) > 0.01 or abs(gyro.z) > 0.01:
                print("  ❌ Gyroscope needs calibration (rotation drift)")
                needs_calibration = True

            if not needs_calibration:
                print("  ✅ IMU appears calibrated")
            else:
                print("\n  🔧 CALIBRATION REQUIRED: Run IMU calibration to fix biases!")

            print("="*60)

    def quaternion_to_euler(self, x, y, z, w):
        """Convert quaternion to Euler angles (roll, pitch, yaw)"""
        # Roll (x-axis rotation)
        sinr_cosp = 2 * (w * x + y * z)
        cosr_cosp = 1 - 2 * (x * x + y * y)
        roll = math.atan2(sinr_cosp, cosr_cosp)

        # Pitch (y-axis rotation)
        sinp = 2 * (w * y - z * x)
        if abs(sinp) >= 1:
            pitch = math.copysign(math.pi / 2, sinp)
        else:
            pitch = math.asin(sinp)
        # Yaw (z-axis rotation)
        siny_cosp = 2 * (w * z + x * y)
        cosy_cosp = 1 - 2 * (y * y + z * z)
        yaw = math.atan2(siny_cosp, cosy_cosp)

        # Convert to degrees
        roll_deg = math.degrees(roll)
        pitch_deg = math.degrees(pitch)
        yaw_deg = math.degrees(yaw)

        return roll_deg, pitch_deg, yaw_deg

def main(args=None):
    rclpy.init(args=args)
    imu_display = IMUDisplay()

    try:
        rclpy.spin(imu_display)
    except KeyboardInterrupt:
        print("\n" + "="*60)
        print("IMU DATA DISPLAY STOPPED")
        print("Total samples received:
{}".format(imu_display.sample_count))
        print("="*60)
    finally:
        imu_display.destroy_node()
        rclpy.shutdown()


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

Purpose: Displays IMU data in terminal and shows calibration requirements.

File 4: Robot Mover Node

demo_sensor/robot_mover.py

```
python
```

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


import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
import time


class RobotMover(Node):
    def __init__(self):
        super().__init__('robot_mover')


        # Publisher for robot velocity commands
        self.publisher = self.create_publisher(Twist, '/cmd_vel', 10)


        # Timer to send commands periodically (10 Hz)
        self.timer = self.create_timer(0.1, self.move_robot)
        self.start_time = self.get_clock().now()
        self.step = 0

        self.get_logger().info('🤖 ROBOT MOVER STARTED')
        self.get_logger().info('Robot will move automatically to demonstrate IMU data changes')

    def move_robot(self):
        current_time = self.get_clock().now()
        elapsed_time = (current_time - self.start_time).nanoseconds / 1e9

        msg = Twist()

        # Movement pattern: Forward -> Turn Left -> Forward -> Turn Right -> Stop -> Repeat
        if elapsed_time < 3.0:
            # Move forward for 3 seconds
            msg.linear.x = 0.3
            msg.angular.z = 0.0
            if self.step != 1:
                self.get_logger().info('🚀 Moving FORWARD - Watch acceleration in IMU data!')
                self.step = 1

        elif elapsed_time < 6.0:
            # Turn left for 3 seconds
            msg.linear.x = 0.0
            msg.angular.z = 0.5
            if self.step != 2:
                self.get_logger().info('🔄 Turning LEFT - Watch gyroscope Z-axis!')
                self.step = 2

        elif elapsed_time < 9.0:
            # Move forward for 3 seconds
            msg.linear.x = 0.3
            msg.angular.z = 0.0
            if self.step != 3:
                self.get_logger().info('🚀 Moving FORWARD - Acceleration changes again!')
                self.step = 3

        elif elapsed_time < 12.0:
            # Turn right for 3 seconds
            msg.linear.x = 0.0
            msg.angular.z = -0.5
            if self.step != 4:
                self.get_logger().info('🔄 Turning RIGHT - Negative gyroscope Z-axis!')
                self.step = 4

        elif elapsed_time < 15.0:
            # Stop for 3 seconds
            msg.linear.x = 0.0
            msg.angular.z = 0.0
            if self.step != 5:
                self.get_logger().info('🛑 STOPPING - IMU should stabilize (check for biases!)')
                self.step = 5

        else:
            # Reset the cycle
            self.start_time = self.get_clock().now()
            self.step = 0
            self.get_logger().info('🔄 Restarting movement cycle...')

        self.publisher.publish(msg)

def main(args=None):
    rclpy.init(args=args)
    robot_mover = RobotMover()

    try:
        rclpy.spin(robot_mover)
    except KeyboardInterrupt:
        # Stop the robot before exiting
        stop_msg = Twist()
        robot_mover.publisher.publish(stop_msg)
        robot_mover.get_logger().info('🛑 Robot stopped and shutting down...')
    finally:
        robot_mover.destroy_node()
        rclpy.shutdown()


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

Purpose: Automatically moves robot to generate IMU data for observation.

File 5: Launch File

launch/simulation.launch.py

```python
from launch import LaunchDescription
from launch.actions import ExecuteProcess, TimerAction
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory
import os


def generate_launch_description():
    package_name = 'demo_sensor'
    package_share = get_package_share_directory(package_name)

    # Robot URDF file
    urdf_file = os.path.join(package_share, 'urdf', 'simple_robot.urdf.xacro')

    # World file (use default empty world)
    world_file = os.path.join(get_package_share_directory('gazebo_ros'),
'worlds', 'empty.world')

    # Robot State Publisher Node - Publishes robot transforms
    robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        arguments=[urdf_file]
    )

    # Spawn Entity Node - Spawns robot in Gazebo
    spawn_entity = Node(
        package='gazebo_ros',
        executable='spawn_entity.py',
        arguments=['-entity', 'simple_robot', '-topic',
'robot_description', '-x', '0.0', '-y', '0.0', '-z', '0.1'],
        output='screen'
    )

    # Gazebo Process - Starts Gazebo simulator
    gazebo = ExecuteProcess(
        cmd=['gazebo', '--verbose', world_file, '-s',
'libgazebo_ros_init.so',
             '-s', 'libgazebo_ros_factory.so'],
        output='screen'
```

```
)

# IMU Display Node - Shows IMU data in terminal
imu_display = Node(
    package=package_name,
    executable='imu_display',
    name='imu_display',
    output='screen'
)

# Robot Mover Node - Moves robot automatically
robot_mover = Node(
    package=package_name,
    executable='robot_mover',
    name='robot_mover',
    output='screen'
)

# Start IMU display early to see data immediately
immediate_imu_display = TimerAction(
    period=1.0,  # Start after 1 second
    actions=[imu_display]
)

# Start robot mover after 3 seconds
delayed_robot_mover = TimerAction(
    period=3.0,
    actions=[robot_mover]
)

return LaunchDescription([
    # Start these immediately
    gazebo,
    robot_state_publisher,
    spawn_entity,

    # Start IMU display early
    immediate_imu_display,

    # Start robot mover later
    delayed_robot_mover,
])
```

Purpose: Coordinates all nodes and starts them in proper order.

Setup and Run Commands

1. Build the Package

bash

```
cd ~/ros2_ws
colcon build --packages-select demo_sensor
source install/setup.bash
```

2. Run the Simulation

```bash
ros2 launch demo_sensor simulation.launch.py
```

Expected Output in Terminal

```text
=== IMU DATA DISPLAY STARTED ===

Waiting for IMU data...

When robot is STATIONARY, expect:

  Acceleration: [0.0, 0.0, ~9.8] m/s²

  Gyroscope:    [0.0, 0.0, 0.0] rad/s

Any non-zero gyro or wrong accel values indicate CALIBRATION NEEDED!


✓ IMU DATA RECEIVED! Showing real-time data:


============================================================

IMU SENSOR DATA - Sample #50

============================================================

ACCELERATION (m/s²):

  X:   0.1234 | Y:  -0.0456 | Z:   9.8123

  ⚠️  X/Y should be ~0.0 when stationary - CALIBRATION NEEDED!


GYROSCOPE (rad/s):

  X:   0.0023 | Y:  -0.0018 | Z:   0.0009

  ⚠️  Should be [0,0,0] when stationary - CALIBRATION NEEDED!
```

```
ORIENTATION (degrees):

  Roll:   0.5° | Pitch:  -1.2° | Yaw:  45.3°


CALIBRATION STATUS:

  ❌ Accelerometer needs calibration (X/Y bias)

  ❌ Gyroscope needs calibration (rotation drift)


  🔧 CALIBRATION REQUIRED: Run IMU calibration to fix biases!
============================================================
```

1.2 LiDAR (Light Detection and Ranging) Fundamentals

What is LiDAR?

LiDAR uses laser beams to measure distances to objects, creating a 360° point cloud of the environment.

Key Concepts:

- Scanning Principle: Rotating laser measures distance at each angle
- Point Cloud: Collection of distance measurements forming an environmental map
- Range and Resolution: Minimum/maximum distance and angular step size
- Beam Characteristics: Laser divergence, intensity, and noise patterns

Data Interpretation:

- Range Values: Distance to obstacles at each angle
- Intensity Data: Reflectivity of surfaces (optional)
- Scan Patterns: Characteristic patterns for walls, corners, open spaces
- Obstacle Detection: Identifying objects based on range discontinuities

Environmental Analysis:

- Open Space: Sparse valid readings, large maximum ranges

- Corridor: Parallel walls at consistent distances
- Cluttered Environment: Many close-range readings, complex patterns
- Dynamic Objects: Changing range patterns over time

LiDAR Basics & Data Visualization

Objective

Set up a robot with LiDAR sensor in Gazebo, visualize LiDAR data in terminal and RViz, and understand LiDAR scanning principles.

Updated Project Structure

```text
demo_sensor/
├── package.xml
├── setup.py
├── setup.cfg
├── launch/
│   └── simulation.launch.py
├── urdf/
│   └── simple_robot.urdf.xacro
└── demo_sensor/
    ├── __init__.py
    ├── imu_display.py
    ├── robot_mover.py
    ├── lidar_display.py
    └── lidar_visualizer.py
```

File 1: Updated Package Configuration

package.xml (add laser geometry dependencies)

xml

```xml
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypelayout="1.0"?>
<package format="3">
  <name>demo_sensor</name>
  <version>0.1.0</version>
  <description>IMU and LiDAR Sensor Demonstration</description>
  <maintainer email="user@example.com">User</maintainer>
  <license>Apache-2.0</license>

  <depend>rclpy</depend>
  <depend>gazebo_ros</depend>
  <depend>geometry_msgs</depend>
  <depend>sensor_msgs</depend>
  <depend>robot_state_publisher</depend>
  <depend>laser_geometry</depend>
  <depend>tf2_ros</depend>
  <depend>visualization_msgs</depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
```

```
</package>
```

setup.py (add LiDAR nodes)

```
python
```

```
from setuptools import setup


package_name = 'demo_sensor'


setup(
    name=package_name,
    version='0.1.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        ('share/' + package_name + '/launch',
['launch/simulation.launch.py']),
        ('share/' + package_name + '/urdf',
['urdf/simple_robot.urdf.xacro']),
        ('share/' + package_name + '/rviz', ['rviz/sensors_demo.rviz']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='User',
    maintainer_email='user@example.com',
    description='IMU and LiDAR Sensor Demonstration',
    license='Apache-2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'imu_display = demo_sensor.imu_display:main',
            'robot_mover = demo_sensor.robot_mover:main',
            'lidar_display = demo_sensor.lidar_display:main',
            'lidar_visualizer = demo_sensor.lidar_visualizer:main',
        ],
    },
)
```

File 2: Updated Robot URDF with LiDAR

urdf/simple_robot.urdf.xacro (add LiDAR components)

xml

```xml
<?xml version="1.0"?>
<robot name="simple_robot" xmlns:xacro="http://www.ros.org/wiki/xacro">

  <!-- Base Link - Main body of the robot -->
  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.4 0.2 0.1"/>
      </geometry>
      <material name="blue">
```

```xml
        <color rgba="0 0 0.8 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <box size="0.4 0.2 0.1"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="2.0"/>
      <inertia ixx="0.1" ixy="0.0" ixz="0.0" iyy="0.1" iyz="0.0"
izz="0.1"/>
    </inertial>
  </link>

  <!-- IMU Link - Red box representing IMU sensor -->
  <link name="imu_link">
    <visual>
      <geometry>
        <box size="0.05 0.05 0.02"/>
      </geometry>
      <material name="red">
        <color rgba="1 0 0 1"/>
      </material>
    </visual>
  </link>
```

```
<!-- LiDAR Link - Cylinder representing LiDAR sensor -->
<link name="lidar_link">
  <visual>
    <geometry>
      <cylinder length="0.03" radius="0.04"/>
    </geometry>
    <material name="green">
      <color rgba="0 1 0 1"/>
    </material>
  </visual>
</link>

<!-- IMU Joint - Fixed connection to base -->
<joint name="imu_joint" type="fixed">
  <parent link="base_link"/>
  <child link="imu_link"/>
  <origin xyz="0.1 0.0 0.05" rpy="0 0 0"/>
</joint>

<!-- LiDAR Joint - Fixed connection to base -->
<joint name="lidar_joint" type="fixed">
  <parent link="base_link"/>
  <child link="lidar_link"/>
  <origin xyz="0.0 0.0 0.08" rpy="0 0 0"/>
</joint>
```

```xml
  <!-- Left Wheel -->
  <link name="left_wheel">
    <visual>
      <geometry>
        <cylinder length="0.05" radius="0.06"/>
      </geometry>
      <material name="black">
        <color rgba="0 0 0 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.05" radius="0.06"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.5"/>
      <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0" izz="0.001"/>
    </inertial>
  </link>

  <!-- Right Wheel -->
  <link name="right_wheel">
    <visual>
```

```
      <geometry>
        <cylinder length="0.05" radius="0.06"/>
      </geometry>
      <material name="black">
        <color rgba="0 0 0 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.05" radius="0.06"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.5"/>
      <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0"
izz="0.001"/>
    </inertial>
  </link>

  <!-- Caster Wheel -->
  <link name="caster_wheel">
    <visual>
      <geometry>
        <sphere radius="0.03"/>
      </geometry>
      <material name="gray">
```

```
        <color rgba="0.5 0.5 0.5 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <sphere radius="0.03"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0" izz="0.0001"/>
    </inertial>
  </link>

  <!-- Wheel Joints -->
  <joint name="left_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="left_wheel"/>
    <origin xyz="0.0 0.1 -0.025" rpy="1.5708 0 0"/>
    <axis xyz="0 0 1"/>
  </joint>

  <joint name="right_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="right_wheel"/>
    <origin xyz="0.0 -0.1 -0.025" rpy="1.5708 0 0"/>
    <axis xyz="0 0 1"/>
  </joint>

  <joint name="caster_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="caster_wheel"/>
    <origin xyz="-0.15 0.0 -0.05" rpy="0 0 0"/>
    <axis xyz="0 0 1"/>
  </joint>

  <!-- Gazebo Plugins -->
  <gazebo>
    <!-- Differential Drive Controller - Enables robot movement -->
    <plugin name="diff_drive_controller"
filename="libgazebo_ros_diff_drive.so">
      <command_topic>/cmd_vel</command_topic>
      <odometry_topic>/odom</odometry_topic>
      <odometry_frame>odom</odometry_frame>
      <robot_base_frame>base_link</robot_base_frame>
      <left_joint>left_wheel_joint</left_joint>
      <right_joint>right_wheel_joint</right_joint>
      <wheel_separation>0.2</wheel_separation>
      <wheel_diameter>0.12</wheel_diameter>
      <publish_odom>true</publish_odom>
      <publish_odom_tf>true</publish_odom_tf>
      <publish_wheel_tf>true</publish_wheel_tf>

      <max_wheel_torque>20</max_wheel_torque>

      <wheel_acceleration>1.0</wheel_acceleration>

    </plugin>

  </gazebo>


  <!-- IMU Sensor Plugin - Publishes IMU data to /imu/data topic -->

  <gazebo reference="imu_link">

    <sensor name="imu_sensor" type="imu">

      <always_on>true</always_on>

      <update_rate>100</update_rate>

      <imu>

        <topic>/imu/data</topic>

        <noise>

          <type>gaussian</type>

          <rate>

            <mean>0.0</mean>

            <stddev>0.0002</stddev>

            <bias_mean>0.0</bias_mean>

            <bias_stddev>0.00003</bias_stddev>

          </rate>

          <accel>

            <mean>0.0</mean>

            <stddev>0.017</stddev>

            <bias_mean>0.1</bias_mean>

            <bias_stddev>0.001</bias_stddev>
```

```xml
          </accel>
        </noise>
      </imu>
    </sensor>
  </gazebo>

  <!-- LiDAR Sensor Plugin - Publishes laser scan data to /scan topic -->
  <gazebo reference="lidar_link">
    <sensor name="lidar_sensor" type="ray">
      <always_on>true</always_on>
      <update_rate>10</update_rate>
      <visualize>true</visualize>
      <ray>
        <scan>
          <horizontal>
            <samples>360</samples>       <!-- 360 samples for 360° coverage -->
            <resolution>1.0</resolution> <!-- 1° resolution -->
            <min_angle>-3.14159</min_angle> <!-- -180° -->
            <max_angle>3.14159</max_angle>  <!-- +180° -->
          </horizontal>
        </scan>
        <range>
          <min>0.08</min>     <!-- 8cm minimum range -->
          <max>10.0</max>     <!-- 10m maximum range -->
          <resolution>0.01</resolution> <!-- 1cm resolution -->
        </range>
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.01</stddev> <!-- 1cm noise -->
        </noise>
      </ray>
      <plugin name="lidar_controller"
filename="libgazebo_ros_ray_sensor.so">
        <topic_name>/scan</topic_name>        <!-- LaserScan topic -->
        <frame_name>lidar_link</frame_name>   <!-- TF frame -->
        <radiation>laser</radiation>
      </plugin>
    </sensor>
  </gazebo>


  <!-- Gazebo Material Colors -->
  <gazebo reference="base_link">
    <material>Gazebo/Blue</material>
  </gazebo>
  <gazebo reference="imu_link">
    <material>Gazebo/Red</material>
  </gazebo>
  <gazebo reference="lidar_link">
    <material>Gazebo/Green</material>
  </gazebo>
  <gazebo reference="left_wheel">
    <material>Gazebo/Black</material>
  </gazebo>
  <gazebo reference="right_wheel">
    <material>Gazebo/Black</material>
  </gazebo>
  <gazebo reference="caster_wheel">
    <material>Gazebo/Gray</material>
  </gazebo>

</robot>
```

Purpose: Adds LiDAR sensor with 360° scanning capability and configures Gazebo plugin.

File 3: LiDAR Data Display Node

demo_sensor/lidar_display.py

```
python
```

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

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan
import math
import numpy as np
```

```
class LiDARDisplay(Node):
    def __init__(self):
        super().__init__('lidar_display')

        # Subscribe to LiDAR scan data
        self.subscription = self.create_subscription(
            LaserScan,
            '/scan',
            self.lidar_callback,
            10)

        self.get_logger().info('=== LiDAR DATA DISPLAY STARTED ===')
        self.get_logger().info('Waiting for LiDAR data...')
        self.get_logger().info('LiDAR Specifications:')
        self.get_logger().info('  • 360° scanning range')
        self.get_logger().info('  • 1° angular resolution')
        self.get_logger().info('  • 8cm to 10m range')
        self.get_logger().info('  • 360 samples per scan')

        self.scan_count = 0
        self.last_ranges = None

    def lidar_callback(self, msg):
        # Show first scan immediately
        if self.scan_count == 0:
            self.get_logger().info('✓ LiDAR DATA RECEIVED! Showing scan information:')
            self.print_scan_info(msg)

        self.scan_count += 1

        # Only print every 10 scans to avoid spam
        if self.scan_count % 10 == 0:
            self.analyze_scan(msg)

    def print_scan_info(self, msg):
        """Print LiDAR scan specifications"""
        print("\n" + "="*70)
        print("LiDAR SCAN SPECIFICATIONS")
        print("="*70)
        print(f"Scan angle: {math.degrees(msg.angle_min):.1f}° to {math.degrees(msg.angle_max):.1f}°")
        print(f"Angular resolution: {math.degrees(msg.angle_increment):.2f}°")
        print(f"Range: {msg.range_min:.2f}m to {msg.range_max:.2f}m")
        print(f"Samples per scan: {len(msg.ranges)}")
        print(f"Scan time: {msg.scan_time:.3f}s")
        print(f"Time increment: {msg.time_increment:.6f}s")
        print("="*70)

    def analyze_scan(self, msg):
        """Analyze and display current scan data"""
        ranges = np.array(msg.ranges)

        # Filter out invalid readings (inf, nan, out of range)
        valid_ranges = ranges[(ranges > msg.range_min) & (ranges < msg.range_max)]

        if len(valid_ranges) > 0:
            min_range = np.min(valid_ranges)
            max_range = np.max(valid_ranges)
            avg_range = np.mean(valid_ranges)

            # Find closest object direction
            min_idx = np.argmin(ranges)
            min_angle_deg = math.degrees(msg.angle_min + min_idx * msg.angle_increment)

            # Count obstacles at different distances
            close_obstacles = np.sum(valid_ranges < 1.0)  # < 1m
            mid_obstacles = np.sum((valid_ranges >= 1.0) & (valid_ranges < 3.0))  # 1-3m
            far_obstacles = np.sum(valid_ranges >= 3.0)  # >= 3m

            print("\n" + "="*70)
            print(f"LiDAR SCAN ANALYSIS - Scan #{self.scan_count}")
            print("="*70)
            print(f"Valid readings: {len(valid_ranges)}/{len(ranges)} ({len(valid_ranges)/len(ranges)*100:.1f}%)")
            print(f"Range statistics: Min={min_range:.2f}m, Max={max_range:.2f}m, Avg={avg_range:.2f}m")
            print(f"Closest object: {min_range:.2f}m at {min_angle_deg:.1f}°")
            print(f"Obstacle distribution: Close(<1m)={close_obstacles}, Mid(1-3m)={mid_obstacles}, Far(>3m)={far_obstacles}")

            # Detect if robot is in open space, corridor, or surrounded
            if len(valid_ranges) < 100:
                environment = "OPEN SPACE"
            elif close_obstacles > 50:
                environment = "SURROUNDED"
            elif abs(min_angle_deg) < 45 or abs(min_angle_deg) > 135:
                environment = "CORRIDOR (front/back)"
            else:
                environment = "CORRIDOR (sides)"

            print(f"Environment: {environment}")

            # Show detection sectors
            front = ranges[0:30] + ranges[-30:]  # -30° to +30°
            left = ranges[60:120]                # 60° to 120°
            right = ranges[240:300]              # 240° to 300°
            back = ranges[150:210]               # 150° to 210°

            front_min = np.min(front[front < msg.range_max]) if np.any(front < msg.range_max) else float('inf')
            left_min = np.min(left[left < msg.range_max]) if np.any(left < msg.range_max) else float('inf')
            right_min = np.min(right[right < msg.range_max]) if np.any(right < msg.range_max) else float('inf')
            back_min = np.min(back[back < msg.range_max]) if np.any(back < msg.range_max) else float('inf')

            print("\nSECTOR DETECTION:")
            print(f"  Front: {front_min:.2f}m | Left: {left_min:.2f}m | Right: {right_min:.2f}m | Back: {back_min:.2f}m")

            # Navigation suggestions
            print("\nNAVIGATION SUGGESTIONS:")
            if front_min < 0.5:
                print("  ⚠️  OBSTACLE AHEAD! Turn left or right")
            elif front_min < 1.0:
                print("  ⚠️  Object approaching in front - Slow down")
            else:
                print("  ✅ Clear path ahead")

            if left_min < right_min and left_min > 0.5:
                print("  ↩ Better to turn left (more space)")
            elif right_min < left_min and right_min > 0.5:
                print("  ↪ Better to turn right (more space)")

            print("="*70)
            self.last_ranges = ranges


def main(args=None):
    rclpy.init(args=args)
    lidar_display = LiDARDisplay()

    try:
        rclpy.spin(lidar_display)
    except KeyboardInterrupt:
        print("\n" + "="*70)
        print("LiDAR DATA DISPLAY STOPPED")
        print(f"Total scans analyzed: {lidar_display.scan_count}")
        print("="*70)
    finally:
        lidar_display.destroy_node()
        rclpy.shutdown()


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

Purpose: Displays LiDAR scan data with environmental analysis and navigation suggestions.

File 4: LiDAR Visualizer Node

demo_sensor/lidar_visualizer.py

```
python
```

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

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan
from visualization_msgs.msg import Marker, MarkerArray
from geometry_msgs.msg import Point
from std_msgs.msg import ColorRGBA
import math

class LiDARVisualizer(Node):
    def __init__(self):
        super().__init__('lidar_visualizer')

        # Subscribe to LiDAR data
        self.subscription = self.create_subscription(
            LaserScan,
            '/scan',
            self.lidar_callback,
            10)

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

        self.get_logger().info('LiDAR Visualizer started - Creating RViz visualization')
    def lidar_callback(self, msg):
        marker_array = MarkerArray()

        # Create text marker showing scan info
        text_marker = Marker()
        text_marker.header.frame_id = "base_link"
        text_marker.header.stamp = self.get_clock().now().to_msg()
        text_marker.ns = "lidar_info"
        text_marker.id = 0
        text_marker.type = Marker.TEXT_VIEW_FACING
        text_marker.action = Marker.ADD

        text_marker.pose.position.x = 0.0
        text_marker.pose.position.y = 0.0
        text_marker.pose.position.z = 1.0

        text_marker.scale.z = 0.1
        text_marker.color.a = 1.0
        text_marker.color.r = 1.0
        text_marker.color.g = 1.0
        text_marker.color.b = 1.0

        # Count valid readings
        valid_ranges = [r for r in msg.ranges if msg.range_min < r <
msg.range_max]
```

```
        text_marker.text = f"LiDAR
Scan\n{len(valid_ranges)}/{len(msg.ranges)}
valid\n{len(valid_ranges)/len(msg.ranges)*100:.1f}% coverage"

        marker_array.markers.append(text_marker)

        # Create point markers for detected obstacles
        point_marker = Marker()
        point_marker.header.frame_id = "lidar_link"
        point_marker.header.stamp = self.get_clock().now().to_msg()
        point_marker.ns = "lidar_points"
        point_marker.id = 1
        point_marker.type = Marker.POINTS
        point_marker.action = Marker.ADD

        point_marker.scale.x = 0.05
        point_marker.scale.y = 0.05
        point_marker.color.a = 0.8
        point_marker.color.r = 0.0
        point_marker.color.g = 1.0
        point_marker.color.b = 0.0

        # Add points for valid range readings
        for i, range_val in enumerate(msg.ranges):
            if msg.range_min < range_val < msg.range_max:
                angle = msg.angle_min + i * msg.angle_increment
                x = range_val * math.cos(angle)
                y = range_val * math.sin(angle)

                point = Point()
                point.x = x
                point.y = y
                point.z = 0.0

                point_marker.points.append(point)

        marker_array.markers.append(point_marker)

        # Create closest obstacle marker
        closest_range = min([r for r in msg.ranges if msg.range_min < r <
msg.range_max], default=None)
        if closest_range:
            closest_idx = msg.ranges.index(closest_range)
            closest_angle = msg.angle_min + closest_idx *
msg.angle_increment

            closest_marker = Marker()
            closest_marker.header.frame_id = "lidar_link"
            closest_marker.header.stamp = self.get_clock().now().to_msg()
            closest_marker.ns = "closest_obstacle"
            closest_marker.id = 2
            closest_marker.type = Marker.SPHERE
            closest_marker.action = Marker.ADD
            closest_marker.pose.position.x = closest_range *
math.cos(closest_angle)
            closest_marker.pose.position.y = closest_range *
math.sin(closest_angle)
            closest_marker.pose.position.z = 0.0

            closest_marker.scale.x = 0.1
            closest_marker.scale.y = 0.1
            closest_marker.scale.z = 0.1

            closest_marker.color.a = 1.0
            closest_marker.color.r = 1.0
            closest_marker.color.g = 0.0
            closest_marker.color.b = 0.0

            marker_array.markers.append(closest_marker)

        self.marker_pub.publish(marker_array)

def main(args=None):
    rclpy.init(args=args)
    lidar_visualizer = LiDARVisualizer()

    try:
        rclpy.spin(lidar_visualizer)
    except KeyboardInterrupt:
        lidar_visualizer.get_logger().info('LiDAR Visualizer stopped')
    finally:
        lidar_visualizer.destroy_node()
        rclpy.shutdown()


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

Purpose: Creates RViz visualization for LiDAR data including detected points and closest obstacle.

File 5: Updated Launch File

launch/simulation.launch.py

python

```
from launch import LaunchDescription
from launch.actions import ExecuteProcess, TimerAction
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory
import os


def generate_launch_description():
    package_name = 'demo_sensor'
    package_share = get_package_share_directory(package_name)


    # Robot URDF file
    urdf_file = os.path.join(package_share, 'urdf', 'simple_robot.urdf.xacro')
    # World file (use default empty world)
    world_file = os.path.join(get_package_share_directory('gazebo_ros'), 'worlds', 'empty.world')

    # RViz config file
    rviz_config = os.path.join(package_share, 'rviz', 'sensors_demo.rviz')

    # Robot State Publisher Node - Publishes robot transforms
    robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        arguments=[urdf_file]
    )

    # Spawn Entity Node - Spawns robot in Gazebo
    spawn_entity = Node(
        package='gazebo_ros',
        executable='spawn_entity.py',
        arguments=['-entity', 'simple_robot', '-topic', 'robot_description', '-x', '0.0', '-y', '0.0', '-z', '0.1'],
        output='screen'
    )

    # Gazebo Process - Starts Gazebo simulator
    gazebo = ExecuteProcess(
        cmd=['gazebo', '--verbose', world_file, '-s',
'libgazebo_ros_init.so',
             '-s', 'libgazebo_ros_factory.so'],
        output='screen'
    )

    # IMU Display Node - Shows IMU data in terminal
    imu_display = Node(
        package=package_name,
        executable='imu_display',
        name='imu_display',
        output='screen'
    )

    # LiDAR Display Node - Shows LiDAR data in terminal
    lidar_display = Node(
        package=package_name,
        executable='lidar_display',
        name='lidar_display',
        output='screen'
    )

    # LiDAR Visualizer Node - Creates RViz visualization
    lidar_visualizer = Node(
        package=package_name,
        executable='lidar_visualizer',
        name='lidar_visualizer',
        output='screen'
    )

    # Robot Mover Node - Moves robot automatically
    robot_mover = Node(
        package=package_name,
        executable='robot_mover',
        name='robot_mover',
        output='screen'
    )

    # RViz2 Node - Visualization tool
    rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        name='rviz2',
        arguments=['-d', rviz_config],
        output='screen'
    )

    # Start sensor displays early
    immediate_sensors = TimerAction(
        period=1.0,
        actions=[imu_display, lidar_display, lidar_visualizer]
```

```
)

# Start robot mover after 3 seconds
delayed_robot_mover = TimerAction(
    period=3.0,
    actions=[robot_mover]
)

# Start RViz after 5 seconds
delayed_rviz = TimerAction(
    period=5.0,
    actions=[rviz_node]
)

return LaunchDescription([
    # Start these immediately
    gazebo,
    robot_state_publisher,
    spawn_entity,

    # Start sensor displays early
    immediate_sensors,

    # Start robot mover and RViz later
    delayed_robot_mover,
    delayed_rviz,
    ])
```

File 6: RViz Configuration

rviz/sensors_demo.rviz (basic configuration)

```yaml
Panels:
  - Class: rviz_common/Displays
    Help Height: 78
    Name: Displays
    Property Tree Widget:
      Expanded:
        - /Global Options1
        - /TF1
      Splitter Ratio: 0.5
    Tree Height: 526
  - Class: rviz_common/Selection
    Name: Selection
  - Class: rviz_common/Tool Properties
    Expanded:
      - /2D Goal Pose1
      - /Publish Point1
    Name: Tool Properties
    Splitter Ratio: 0.5886790156364441
  - Class: rviz_common/Views
    Expanded:
      - /Current View1
    Name: Views
    Splitter Ratio: 0.5
Visualization Manager:
  Class: ""
  Displays:
    - Alpha: 0.5
      Cell Size: 1
      Class: rviz_default_plugins/Grid
      Color: 160; 160; 164
      Enabled: true
      Line Style:
        Line Width: 0.029999999329447746
        Value: Lines
      Name: Grid
      Normal Cell Count: 0
      Offset:
        X: 0
        Y: 0
        Z: 0
      Plane: XY
      Plane Cell Count: 10
      Reference Frame: odom
      Value: true
    - Class: rviz_default_plugins/TF
      Enabled: true
      Frame Timeout: 15
      Frames:
        All Enabled: true
      Marker Scale: 1.0
      Name: TF
      Show Arrows: true
      Show Axes: true
      Show Names: true
      Update Interval: 0
      Value: true
    - Class: rviz_default_plugins/RobotModel
      Description Topic:
        Depth: 5
        Durability Policy: Volatile
        History Policy: Keep Last
        Reliability Policy: Reliable
        Value: /robot_description
      Enabled: true
      Name: RobotModel
      Value: true
    - Class: rviz_default_plugins/LaserScan
      Enabled: true
      Name: LiDAR Scan
      Topic:
        Depth: 5
        Durability Policy: Volatile
```

```yaml
      History Policy: Keep Last
      Reliability Policy: Reliable
      Value: /scan
    Value: true
  - Class: rviz_default_plugins/MarkerArray
    Enabled: true
    Name: LiDAR Visualization
    Topic:
      Depth: 5
      Durability Policy: Volatile
      History Policy: Keep Last
      Reliability Policy: Reliable
      Value: /lidar/markers
    Value: true
Enabled: true
Global Options:
  Background Color: 48; 48; 48
  Fixed Frame: odom
  Frame Rate: 30
Name: root
Toolbars:
  name: Toolbars
Value: true
Views:
  Current:
    Class: rviz_default_plugins/Orbit
```

```
      Distance: 5.0
      Enable Stereo Rendering:
        Stereo Eye Separation: 0.05999999865889549
        Stereo Focal Distance: 1.0
        Swap Stereo Eyes: false
        Value: false
      Focal Point:
        X: 0.0
        Y: 0.0
        Z: 0.0
      Focal Shape Fixed Size: true
      Focal Shape Size: 0.05000000074505806
      Name: Current View
      Near Clip Distance: 0.009999999776482582
      Pitch: 0.5
      Target Frame: <Fixed Frame>
      Value: Orbit (rviz)
      Yaw: 0.0
    Saved: ~
Window Geometry:
  Displays:
    collapsed: false
  Height: 927
  Hide Left Dock: false
  Hide Right Dock: false
  QMainWindow State:
000000ff00000000fd000000040000000000000156000002f4fc0200000008fb00000012005
30065006c0065006300740069006f006e00000001e10000009b0000005c00fffffffb000000
1e0054006f006f006c002000500072006f0070006500720074006900650073020000001ed000
001df00000185000000a3fb000000120056006900650077007300200054006f006f02000001
df000002110000018500000122fb000000200054006f006f006c002000500072006f0070006
500720074006900650073003203000002880000011d000002210000017afb00000010004400
6900730070006c006100790073010000003d000002f4000000c900ffffff
```

Setup and Run Commands

1. Build the Package

```bash
cd ~/ros2_ws
colcon build --packages-select demo_sensor
source install/setup.bash
```

2. Run the Simulation

```bash
ros2 launch demo_sensor simulation.launch.py
```

Expected LiDAR Output in Terminal

```text
=== LiDAR DATA DISPLAY STARTED ===
Waiting for LiDAR data...
LiDAR Specifications:
  • 360° scanning range
  • 1° angular resolution

  • 8cm to 10m range

  • 360 samples per scan


✓ LiDAR DATA RECEIVED! Showing scan information:


======================================================================

LiDAR SCAN SPECIFICATIONS

======================================================================

Scan angle: -180.0° to 180.0°

Angular resolution: 1.00°

Range: 0.08m
```

1.3 Camera Fundamentals

What is a Vision Sensor?

Cameras capture 2D images of the environment, providing rich visual information for object recognition, navigation, and scene understanding.

Key Concepts:

- Image Formation: How 3D world projects to 2D image plane
- Color Spaces: RGB, HSV, and their applications
- Resolution and FOV: Image size and angular coverage
- Exposure and Lighting: Effects of illumination on image quality

Image Analysis Techniques:

- Brightness Analysis: Overall light levels in scene
- Color Detection: Identifying objects by color in HSV space
- Edge Detection: Finding object boundaries and textures

- Contrast Measurement: Scene complexity assessment

Computer Vision Applications:

- Object Recognition: Identifying items by visual features
- Depth Estimation: Inferring distance from visual cues
- Optical Flow: Tracking motion between frames
- Feature Detection: Finding distinctive points in images

Camera Basics & Image Visualization

Objective

Set up a robot with camera sensor in Gazebo, visualize camera data in terminal and RViz, and understand image processing principles.

Updated Project Structure

```text
demo_sensor/
├── package.xml
├── setup.py
├── setup.cfg
├── launch/
│   └── simulation.launch.py
├── urdf/
│   └── simple_robot.urdf.xacro
└── demo_sensor/
    ├── __init__.py
    ├── imu_display.py
    ├── robot_mover.py
    ├── lidar_display.py
    ├── lidar_visualizer.py
    ├── camera_display.py
    └── camera_visualizer.py
```

File 1: Updated Package Configuration

package.xml (add image and camera dependencies)

xml

```xml
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypelayout="1.0"?>
<package format="3">
  <name>demo_sensor</name>
  <version>0.1.0</version>
  <description>IMU, LiDAR and Camera Sensor Demonstration</description>
  <maintainer email="user@example.com">User</maintainer>
  <license>Apache-2.0</license>

  <depend>rclpy</depend>
  <depend>gazebo_ros</depend>
  <depend>geometry_msgs</depend>
  <depend>sensor_msgs</depend>
  <depend>robot_state_publisher</depend>
  <depend>laser_geometry</depend>
  <depend>tf2_ros</depend>
  <depend>visualization_msgs</depend>
  <depend>image_transport</depend>
  <depend>cv_bridge</depend>
  <depend>opencv2</depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

setup.py (add camera nodes)

```python
from setuptools import setup
import os
from glob import glob

package_name = 'demo_sensor'

setup(
    name=package_name,
    version='0.1.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        ('share/' + package_name + '/launch',
['launch/simulation.launch.py']),
        ('share/' + package_name + '/urdf',
['urdf/simple_robot.urdf.xacro']),
        ('share/' + package_name + '/rviz', ['rviz/sensors_demo.rviz']),
        ('share/' + package_name + '/worlds', glob('worlds/*.world')),
    ],
    install_requires=['setuptools', 'opencv-python'],
    zip_safe=True,
    maintainer='User',
    maintainer_email='user@example.com',
    description='IMU, LiDAR and Camera Sensor Demonstration',
    license='Apache-2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'imu_display = demo_sensor.imu_display:main',
            'robot_mover = demo_sensor.robot_mover:main',
            'lidar_display = demo_sensor.lidar_display:main',
            'lidar_visualizer = demo_sensor.lidar_visualizer:main',
            'camera_display = demo_sensor.camera_display:main',
            'camera_visualizer = demo_sensor.camera_visualizer:main',
        ],
    },
)
```

File 2: Updated Robot URDF with Camera

urdf/simple_robot.urdf.xacro (add camera components) xml

```
<?xml version="1.0"?>
<robot name="simple_robot" xmlns:xacro="http://www.ros.org/wiki/xacro">

  <!-- Base Link - Main body of the robot -->
  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.4 0.2 0.1"/>
      </geometry>
      <material name="blue">
        <color rgba="0 0 0.8 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <box size="0.4 0.2 0.1"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="2.0"/>
      <inertia ixx="0.1" ixy="0.0" ixz="0.0" iyy="0.1" iyz="0.0" izz="0.1"/>
    </inertial>
  </link>
```

```
<!-- IMU Link - Red box representing IMU sensor -->
<link name="imu_link">
  <visual>
    <geometry>
      <box size="0.05 0.05 0.02"/>
    </geometry>
    <material name="red">
      <color rgba="1 0 0 1"/>
    </material>
  </visual>
</link>

<!-- LiDAR Link - Cylinder representing LiDAR sensor -->
<link name="lidar_link">
  <visual>
    <geometry>
      <cylinder length="0.03" radius="0.04"/>
    </geometry>
    <material name="green">
      <color rgba="0 1 0 1"/>
    </material>
  </visual>
</link>

<!-- Camera Link - Small box representing camera -->
<link name="camera_link">
```

```xml
  <visual>

    <geometry>

      <box size="0.03 0.05 0.02"/>

    </geometry>

    <material name="yellow">

      <color rgba="1 1 0 1"/>

    </material>

  </visual>

</link>


<!-- IMU Joint - Fixed connection to base -->

<joint name="imu_joint" type="fixed">

  <parent link="base_link"/>

  <child link="imu_link"/>

  <origin xyz="0.1 0.0 0.05" rpy="0 0 0"/>

</joint>


<!-- LiDAR Joint - Fixed connection to base -->

<joint name="lidar_joint" type="fixed">

  <parent link="base_link"/>

  <child link="lidar_link"/>

  <origin xyz="0.0 0.0 0.08" rpy="0 0 0"/>

</joint>


<!-- Camera Joint - Fixed connection to base, facing forward -->

<joint name="camera_joint" type="fixed">
```

```
    <parent link="base_link"/>
    <child link="camera_link"/>
    <origin xyz="0.2 0.0 0.06" rpy="0 0 0"/>
  </joint>

  <!-- Left Wheel -->
  <link name="left_wheel">
    <visual>
      <geometry>
        <cylinder length="0.05" radius="0.06"/>
      </geometry>
      <material name="black">
        <color rgba="0 0 0 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.05" radius="0.06"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.5"/>
      <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0"
izz="0.001"/>
    </inertial>
  </link>
```

```xml
  <!-- Right Wheel -->
  <link name="right_wheel">
    <visual>
      <geometry>
        <cylinder length="0.05" radius="0.06"/>
      </geometry>
      <material name="black">
        <color rgba="0 0 0 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.05" radius="0.06"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.5"/>
      <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0" izz="0.001"/>
    </inertial>
  </link>

  <!-- Caster Wheel -->
  <link name="caster_wheel">
    <visual>
```

```
      <geometry>
        <sphere radius="0.03"/>
      </geometry>
      <material name="gray">
        <color rgba="0.5 0.5 0.5 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <sphere radius="0.03"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0" izz="0.0001"/>
    </inertial>
  </link>

  <!-- Wheel Joints -->
  <joint name="left_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="left_wheel"/>
    <origin xyz="0.0 0.1 -0.025" rpy="1.5708 0 0"/>
    <axis xyz="0 0 1"/>
  </joint>
  <joint name="right_wheel_joint" type="continuous">

    <parent link="base_link"/>

    <child link="right_wheel"/>

    <origin xyz="0.0 -0.1 -0.025" rpy="1.5708 0 0"/>

    <axis xyz="0 0 1"/>

  </joint>


  <joint name="caster_wheel_joint" type="continuous">

    <parent link="base_link"/>

    <child link="caster_wheel"/>

    <origin xyz="-0.15 0.0 -0.05" rpy="0 0 0"/>

    <axis xyz="0 0 1"/>

  </joint>


  <!-- Gazebo Plugins -->

  <gazebo>

    <!-- Differential Drive Controller - Enables robot movement -->

    <plugin name="diff_drive_controller"
filename="libgazebo_ros_diff_drive.so">

      <command_topic>/cmd_vel</command_topic>

      <odometry_topic>/odom</odometry_topic>

      <odometry_frame>odom</odometry_frame>

      <robot_base_frame>base_link</robot_base_frame>

      <left_joint>left_wheel_joint</left_joint>

      <right_joint>right_wheel_joint</right_joint>
      <wheel_separation>0.2</wheel_separation>
      <wheel_diameter>0.12</wheel_diameter>
      <publish_odom>true</publish_odom>
      <publish_odom_tf>true</publish_odom_tf>
      <publish_wheel_tf>true</publish_wheel_tf>
      <max_wheel_torque>20</max_wheel_torque>
      <wheel_acceleration>1.0</wheel_acceleration>
    </plugin>
  </gazebo>

  <!-- IMU Sensor Plugin - Publishes IMU data to /imu/data topic -->
  <gazebo reference="imu_link">
    <sensor name="imu_sensor" type="imu">
      <always_on>true</always_on>
      <update_rate>100</update_rate>
      <imu>
        <topic>/imu/data</topic>
        <noise>
          <type>gaussian</type>
          <rate>
            <mean>0.0</mean>
            <stddev>0.0002</stddev>
            <bias_mean>0.0</bias_mean>
            <bias_stddev>0.00003</bias_stddev>
          </rate>
          <accel>
```

```
          <mean>0.0</mean>
          <stddev>0.017</stddev>
          <bias_mean>0.1</bias_mean>
          <bias_stddev>0.001</bias_stddev>
        </accel>
      </noise>
    </imu>
  </sensor>
</gazebo>

<!-- LiDAR Sensor Plugin - Publishes laser scan data to /scan topic -->
<gazebo reference="lidar_link">
  <sensor name="lidar_sensor" type="ray">
    <always_on>true</always_on>
    <update_rate>10</update_rate>
    <visualize>true</visualize>
    <ray>
      <scan>
        <horizontal>
          <samples>360</samples>
          <resolution>1.0</resolution>
          <min_angle>-3.14159</min_angle>
          <max_angle>3.14159</max_angle>
        </horizontal>
      </scan>
      <range>
```

```
          <min>0.08</min>
          <max>10.0</max>
          <resolution>0.01</resolution>
        </range>
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.01</stddev>
        </noise>
      </ray>
      <plugin name="lidar_controller" filename="libgazebo_ros_ray_sensor.so">
        <topic_name>/scan</topic_name>
        <frame_name>lidar_link</frame_name>
        <radiation>laser</radiation>
      </plugin>
    </sensor>
  </gazebo>

  <!-- Camera Sensor Plugin - Publishes camera images to /camera/image_raw topic -->
  <gazebo reference="camera_link">
    <sensor name="camera_sensor" type="camera">
      <always_on>true</always_on>
      <update_rate>30</update_rate>
      <visualize>true</visualize>
      <camera name="main_camera">
        <horizontal_fov>1.047</horizontal_fov> <!-- 60 degrees -->
        <image>
          <width>640</width>
          <height>480</height>
          <format>R8G8B8</format>
        </image>
        <clip>
          <near>0.05</near>
          <far>100</far>
        </clip>
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.007</stddev>
        </noise>
      </camera>
      <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
        <always_on>true</always_on>
        <update_rate>30</update_rate>
        <camera_name>camera</camera_name>
        <image_topic_name>image_raw</image_topic_name>
        <camera_info_topic_name>camera_info</camera_info_topic_name>
        <frame_name>camera_link</frame_name>
        <hack_baseline>0.07</hack_baseline>
      </plugin>
    </sensor>
```

```xml
  </gazebo>

  <!-- Gazebo Material Colors -->
  <gazebo reference="base_link">
    <material>Gazebo/Blue</material>
  </gazebo>
  <gazebo reference="imu_link">
    <material>Gazebo/Red</material>
  </gazebo>
  <gazebo reference="lidar_link">
    <material>Gazebo/Green</material>
  </gazebo>
  <gazebo reference="camera_link">
    <material>Gazebo/Yellow</material>
  </gazebo>
  <gazebo reference="left_wheel">
    <material>Gazebo/Black</material>
  </gazebo>
  <gazebo reference="right_wheel">
    <material>Gazebo/Black</material>
  </gazebo>
  <gazebo reference="caster_wheel">
    <material>Gazebo/Gray</material>
  </gazebo>

</robot>
```

Purpose: Adds camera sensor with 640x480 resolution and 60° field of view.

File 3: World File with Visual Elements

worlds/sensor_test.world

xml

```
<?xml version="1.0" ?>
<sdf version="1.6">
  <world name="sensor_test">

    <!-- Lighting -->
    <include>
      <uri>model://sun</uri>
    </include>

    <!-- Ground Plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <!-- Colored boxes for camera detection -->
    <model name="red_box">
      <pose>2 0 0.25 0 0 0</pose>
      <link name="link">
        <visual name="visual">
          <geometry>
            <box>
              <size>0.5 0.5 0.5</size>
            </box>
          </geometry>
          <material>
            <ambient>1 0 0 1</ambient>
            <diffuse>1 0 0 1</diffuse>
            <specular>0.1 0.1 0.1 1</specular>
          </material>
        </visual>
        <collision name="collision">
          <geometry>
            <box>
              <size>0.5 0.5 0.5</size>
            </box>
          </geometry>
        </collision>
      </link>
    </model>

    <model name="green_box">
      <pose>1.5 1 0.25 0 0 0</pose>
      <link name="link">
        <visual name="visual">
          <geometry>
            <box>
              <size>0.4 0.4 0.4</size>
            </box>
          </geometry>
          <material>
            <ambient>0 1 0 1</ambient>
            <diffuse>0 1 0 1</diffuse>
            <specular>0.1 0.1 0.1 1</specular>
          </material>
        </visual>
        <collision name="collision">
          <geometry>
            <box>
              <size>0.4 0.4 0.4</size>
            </box>
          </geometry>
        </collision>
      </link>
    </model>

    <model name="blue_box">
      <pose>1.5 -1 0.25 0 0 0</pose>
      <link name="link">
        <visual name="visual">
          <geometry>
            <box>
              <size>0.3 0.3 0.3</size>
            </box>
          </geometry>
          <material>
            <ambient>0 0 1 1</ambient>
            <diffuse>0 0 1 1</diffuse>
            <specular>0.1 0.1 0.1 1</specular>
          </material>
        </visual>
        <collision name="collision">
          <geometry>
            <box>
              <size>0.3 0.3 0.3</size>
            </box>
          </geometry>
        </collision>
      </link>
    </model>

    <!-- Wall for LiDAR and camera testing -->
    <model name="test_wall">
      <pose>3 0 0.5 0 0 0</pose>
      <link name="link">
        <visual name="visual">
          <geometry>
            <box>
              <size>0.1 4 1</size>
            </box>
```

```
          </geometry>
          <material>
            <ambient>0.8 0.8 0.8 1</ambient>
            <diffuse>0.8 0.8 0.8 1</diffuse>
            <specular>0.1 0.1 0.1 1</specular>
          </material>
        </visual>
        <collision name="collision">
          <geometry>
            <box>
              <size>0.1 4 1</size>
            </box>
          </geometry>
        </collision>
      </link>
    </model>

  </world>
</sdf>
```

File 4: Camera Data Display Node

demo_sensor/camera_display.py

python

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

import rclpy
```

```
from rclpy.node import Node
from sensor_msgs.msg import Image, CameraInfo
import cv2
from cv_bridge import CvBridge
import numpy as np

class CameraDisplay(Node):
    def __init__(self):
        super().__init__('camera_display')

        # Subscribe to camera image topic
        self.subscription = self.create_subscription(
            Image,
            '/camera/image_raw',
            self.image_callback,
            10)

        # Subscribe to camera info for specifications
        self.info_subscription = self.create_subscription(
            CameraInfo,
            '/camera/camera_info',
            self.info_callback,
            10)

        self.bridge = CvBridge()
        self.frame_count = 0
        self.camera_info = None

        self.get_logger().info('=== CAMERA DATA DISPLAY STARTED ===')
        self.get_logger().info('Waiting for camera data...')
        self.get_logger().info('Camera will detect:')
        self.get_logger().info('  • Color objects (red, green, blue)')
        self.get_logger().info('  • Brightness levels')
        self.get_logger().info('  • Motion and object detection')

    def info_callback(self, msg):
        """Store camera specifications"""
        if self.camera_info is None:
            self.camera_info = msg
            self.get_logger().info('✓ Camera specifications received')
            self.print_camera_info()

    def print_camera_info(self):
        """Print camera specifications"""
        print("\n" + "="*70)
        print("CAMERA SPECIFICATIONS")
        print("="*70)
        print(f"Image size: {self.camera_info.width} x
{self.camera_info.height}")
        print(f"Focal length: {self.camera_info.k[0]:.1f} (fx),
{self.camera_info.k[4]:.1f} (fy)")
        print(f"Optical center: ({self.camera_info.k[2]:.1f},
{self.camera_info.k[5]:.1f})")
```

```
        print(f"Distortion model: {self.camera_info.distortion_model}")
        print("="*70)

    def image_callback(self, msg):
        """Process and analyze camera images"""
        try:
            # Convert ROS Image message to OpenCV image
            cv_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")
            self.frame_count += 1

            # Only analyze every 30 frames to avoid spam
            if self.frame_count % 30 == 0:
                self.analyze_image(cv_image)

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

    def analyze_image(self, cv_image):
        """Analyze image content and display results"""
        height, width, channels = cv_image.shape

        # Convert to different color spaces for analysis
        hsv = cv2.cvtColor(cv_image, cv2.COLOR_BGR2HSV)
        gray = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)

        # Calculate image statistics
```

```
brightness = np.mean(gray)
contrast = np.std(gray)

# Color detection ranges (HSV)
color_ranges = {
    'RED': ([0, 120, 70], [10, 255, 255]),
    'GREEN': ([36, 100, 100], [86, 255, 255]),
    'BLUE': ([100, 150, 0], [140, 255, 255]),
    'YELLOW': ([20, 100, 100], [30, 255, 255])
}

color_detections = {}

for color_name, (lower, upper) in color_ranges.items():
    # Create mask for color
    lower = np.array(lower, dtype=np.uint8)
    upper = np.array(upper, dtype=np.uint8)
    mask = cv2.inRange(hsv, lower, upper)

    # Count pixels of this color
    pixel_count = np.sum(mask > 0)
    percentage = (pixel_count / (width * height)) * 100

    if percentage > 1.0:  # Only report if significant presence
        color_detections[color_name] = percentage
```

```python
        # Edge detection for object presence
        edges = cv2.Canny(gray, 50, 150)
        edge_density = (np.sum(edges > 0) / (width * height)) * 100

        # Brightness analysis
        brightness_level = "DARK" if brightness < 50 else "NORMAL" if brightness < 150 else "BRIGHT"

        print("\n" + "="*70)
        print(f"CAMERA FRAME ANALYSIS - Frame #{self.frame_count}")
        print("="*70)
        print(f"Image: {width}x{height}, {channels} channels")
        print(f"Brightness: {brightness:.1f} ({brightness_level})")
        print(f"Contrast: {contrast:.1f}")
        print(f"Edge density: {edge_density:.2f}% (indicates texture/complexity)")

        if color_detections:
            print("\nCOLOR DETECTION:")
            for color, percentage in color_detections.items():
                print(f"  {color}: {percentage:.2f}% of image")
        else:
            print("\nCOLOR DETECTION: No significant colors detected")

        # Scene interpretation
        print("\nSCENE INTERPRETATION:")
        if edge_density < 5:
            print("  📷 Simple scene - few edges detected")
        elif edge_density > 20:
            print("  📷 Complex scene - many edges/textures")

        if brightness < 30:
            print("  💡 Low light conditions - consider increasing exposure")
        elif brightness > 200:
            print("  💡 High light conditions - potential overexposure")

        if len(color_detections) >= 2:
            print("  🎨 Multiple colors detected - colorful environment")
        elif not color_detections:
            print("  🎨 Monochromatic scene - limited color variety")

        # Object detection hints based on edges and colors
        if edge_density > 15 and len(color_detections) > 0:
            print("  🔍 Objects likely present in scene")
        elif edge_density < 5:
            print("  🔍 Minimal objects detected - open space")

        print("="*70)

def main(args=None):
    rclpy.init(args=args)
    camera_display = CameraDisplay()
    try:
        rclpy.spin(camera_display)
    except KeyboardInterrupt:
        print("\n" + "="*70)
        print("CAMERA DATA DISPLAY STOPPED")
        print(f"Total frames analyzed: {camera_display.frame_count}")
        print("="*70)
    finally:
        camera_display.destroy_node()
        rclpy.shutdown()


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

Purpose: Analyzes camera images and provides detailed scene analysis.

File 5: Camera Visualizer Node

demo_sensor/camera_visualizer.py

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


import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from visualization_msgs.msg import Marker, MarkerArray
from geometry_msgs.msg import Point
from std_msgs.msg import ColorRGBA
import cv2
from cv_bridge import CvBridge
import numpy as np


class CameraVisualizer(Node):
    def __init__(self):
        super().__init__('camera_visualizer')

        # Subscribe to camera image
        self.subscription = self.create_subscription(
            Image,
            '/camera/image_raw',
            self.image_callback,
            10)

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

        self.bridge = CvBridge()
        self.get_logger().info('Camera Visualizer started - Creating detection visualization')

    def image_callback(self, msg):
        try:
            cv_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")
            self.process_image(cv_image)

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

    def process_image(self, cv_image):
        marker_array = MarkerArray()

        # Convert to HSV for color detection
        hsv = cv2.cvtColor(cv_image, cv2.COLOR_BGR2HSV)
        height, width, _ = cv_image.shape

        # Create text marker with image info
        text_marker = Marker()
        text_marker.header.frame_id = "camera_link"
        text_marker.header.stamp = self.get_clock().now().to_msg()
        text_marker.ns = "camera_info"
        text_marker.id = 0
        text_marker.type = Marker.TEXT_VIEW_FACING
        text_marker.action = Marker.ADD

        text_marker.pose.position.x = 0.0
        text_marker.pose.position.y = 0.0
        text_marker.pose.position.z = 0.3

        text_marker.scale.z = 0.05
        text_marker.color.a = 1.0
        text_marker.color.r = 1.0
        text_marker.color.g = 1.0
        text_marker.color.b = 1.0

        # Calculate basic image stats
        gray = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)
        brightness = np.mean(gray)
        text_marker.text = f"Camera View\n{width}x{height}\nBrightness:
{brightness:.1f}"

        marker_array.markers.append(text_marker)

        # Color detection markers
        colors = {
            'RED': ([0, 120, 70], [10, 255, 255], (1.0, 0.0, 0.0)),
            'GREEN': ([36, 100, 100], [86, 255, 255], (0.0, 1.0, 0.0)),
            'BLUE': ([100, 150, 0], [140, 255, 255], (0.0, 0.0, 1.0)),
        }

        marker_id = 1
        for color_name, (lower, upper, rgb) in colors.items():
            lower = np.array(lower, dtype=np.uint8)
            upper = np.array(upper, dtype=np.uint8)
            mask = cv2.inRange(hsv, lower, upper)
            # Find contours of detected color
            contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)


            for contour in contours:
                if cv2.contourArea(contour) > 100:  # Only significant
areas
                    # Get bounding box
                    x, y, w, h = cv2.boundingRect(contour)


                    # Convert pixel coordinates to 3D space (approximate)
                    # This is a simplified projection - in real systems
you'd use camera intrinsics
                    obj_x = (x + w/2 - width/2) * 0.001  # Convert to
meters
                    obj_y = (y + h/2 - height/2) * 0.001
                    obj_z = 2.0  # Approximate distance


                    # Create bounding box marker
                    bbox_marker = Marker()
                    bbox_marker.header.frame_id = "camera_link"
                    bbox_marker.header.stamp =
self.get_clock().now().to_msg()
                    bbox_marker.ns = f"color_{color_name.lower()}"
                    bbox_marker.id = marker_id
                    bbox_marker.type = Marker.CUBE
                    bbox_marker.action = Marker.ADD
                    bbox_marker.pose.position.x = obj_z  # Distance in front of camera
                    bbox_marker.pose.position.y = -obj_x  # Horizontal position
                    bbox_marker.pose.position.z = -obj_y  # Vertical position
                    bbox_marker.pose.orientation.w = 1.0

                    # Scale based on detected size
                    bbox_marker.scale.x = 0.1
                    bbox_marker.scale.y = w * 0.001
                    bbox_marker.scale.z = h * 0.001

                    bbox_marker.color.a = 0.7
                    bbox_marker.color.r = rgb[0]
                    bbox_marker.color.g = rgb[1]
                    bbox_marker.color.b = rgb[2]

                    marker_array.markers.append(bbox_marker)
                    marker_id += 1

        # Field of view visualization
        fov_marker = Marker()
        fov_marker.header.frame_id = "camera_link"
        fov_marker.header.stamp = self.get_clock().now().to_msg()
        fov_marker.ns = "fov"
        fov_marker.id = 100
        fov_marker.type = Marker.LINE_STRIP
        fov_marker.action = Marker.ADD

        fov_marker.scale.x = 0.02
        fov_marker.color.a = 0.5
        fov_marker.color.r = 1.0
        fov_marker.color.g = 1.0
        fov_marker.color.b = 0.0

        # Create FOV pyramid (simplified)
        points = [
            Point(x=0.0, y=0.0, z=0.0),
            Point(x=2.0, y=1.0, z=0.75),
            Point(x=2.0, y=-1.0, z=0.75),
            Point(x=2.0, y=-1.0, z=-0.75),
            Point(x=2.0, y=1.0, z=-0.75),
            Point(x=2.0, y=1.0, z=0.75),
        ]

        fov_marker.points = points
        marker_array.markers.append(fov_marker)

        self.marker_pub.publish(marker_array)

def main(args=None):
    rclpy.init(args=args)
    camera_visualizer = CameraVisualizer()


    try:

        rclpy.spin(camera_visualizer)

    except KeyboardInterrupt:

        camera_visualizer.get_logger().info('Camera Visualizer stopped')

    finally:

        camera_visualizer.destroy_node()

        rclpy.shutdown()


if __name__ == '__main__':

    main()
```

Purpose: Creates 3D visualization of camera field of view and detected objects.

File 6: Updated Launch File

launch/simulation.launch.py (add camera nodes)

python

```
from launch import LaunchDescription

from launch.actions import ExecuteProcess, TimerAction

from launch_ros.actions import Node

from ament_index_python.packages import get_package_share_directory

import os


def generate_launch_description():

    package_name = 'demo_sensor'

    package_share = get_package_share_directory(package_name)
    # Robot URDF file
    urdf_file = os.path.join(package_share, 'urdf',
'simple_robot.urdf.xacro')

    # World file with visual elements
    world_file = os.path.join(package_share, 'worlds', 'sensor_test.world')

    # RViz config file
    rviz_config = os.path.join(package_share, 'rviz', 'sensors_demo.rviz')

    # Robot State Publisher Node
    robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        arguments=[urdf_file]
    )

    # Spawn Entity Node
    spawn_entity = Node(
        package='gazebo_ros',
        executable='spawn_entity.py',
        arguments=['-entity', 'simple_robot', '-topic',
'robot_description', '-x', '0.0', '-y', '0.0', '-z', '0.1'],
        output='screen'
    )

    # Gazebo Process
    gazebo = ExecuteProcess(
        cmd=['gazebo', '--verbose', world_file, '-s',
'libgazebo_ros_init.so',
             '-s', 'libgazebo_ros_factory.so'],
        output='screen'
    )

    # IMU Display Node
    imu_display = Node(
        package=package_name,
        executable='imu_display',
        name='imu_display',
        output='screen'
    )

    # LiDAR Display Node
    lidar_display = Node(
        package=package_name,
        executable='lidar_display',
        name='lidar_display',
        output='screen'
    )
```

```python
# LiDAR Visualizer Node
lidar_visualizer = Node(
    package=package_name,
    executable='lidar_visualizer',
    name='lidar_visualizer',
    output='screen'
)

# Camera Display Node
camera_display = Node(
    package=package_name,
    executable='camera_display',
    name='camera_display',
    output='screen'
)

# Camera Visualizer Node
camera_visualizer = Node(
    package=package_name,
    executable='camera_visualizer',
    name='camera_visualizer',
    output='screen'
)

# Robot Mover Node
robot_mover = Node(
        package=package_name,
        executable='robot_mover',
        name='robot_mover',
        output='screen'
    )

    # RViz2 Node
    rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        name='rviz2',
        arguments=['-d', rviz_config],
        output='screen'
    )

    # Start sensor displays early
    immediate_sensors = TimerAction(
        period=1.0,
        actions=[imu_display, lidar_display, lidar_visualizer,
camera_display, camera_visualizer]
    )

    # Start robot mover after 3 seconds
    delayed_robot_mover = TimerAction(
        period=3.0,
        actions=[robot_mover]
    )

    # Start RViz after 5 seconds
    delayed_rviz = TimerAction(
        period=5.0,
        actions=[rviz_node]
    )

    return LaunchDescription([
        # Start these immediately
        gazebo,
        robot_state_publisher,
        spawn_entity,

        # Start sensor displays early
        immediate_sensors,

        # Start robot mover and RViz later
        delayed_robot_mover,
        delayed_rviz,
    ])
```

Setup and Run Commands

1. Build the Package

```
bash
```

```
cd ~/ros2_ws

colcon build --packages-select demo_sensor

source install/setup.bash
```

2. Run the Simulation

```
bash

ros2 launch demo_sensor simulation.launch.py
```

Expected Camera Output in Terminal

```
text

=== CAMERA DATA DISPLAY STARTED ===

Waiting for camera data...

Camera will detect:

  • Color objects (red, green, blue)

  • Brightness levels

  • Motion and object detection


✓ Camera specifications received


======================================================================

CAMERA SPECIFICATIONS

======================================================================

Image size: 640 x 480

Focal length: 554.3 (fx), 554.3 (fy)

Optical center: (320.0, 240.0)

Distortion model: plumb_bob
```

```
======================================================================

======================================================================
CAMERA FRAME ANALYSIS - Frame #30
======================================================================
Image: 640x480, 3 channels
Brightness: 127.3 (NORMAL)
Contrast: 45.2
Edge density: 18.53% (indicates texture/complexity)

COLOR DETECTION:
  RED: 12.45% of image
  GREEN: 8.23% of image

SCENE INTERPRETATION:
  📷 Complex scene - many edges/textures
  💡 Normal light conditions
  🎨 Multiple colors detected - colorful environment
  🔍 Objects likely present in scene
======================================================================
```

Learning Objectives

1. Understand Camera Components: Image sensor, lens properties, field of view
2. Learn Image Analysis: Brightness, contrast, color detection, edge detection
3. Object Detection Basics: Using color spaces and contours for object identification

4. Camera Calibration Need: Understand why lens distortion correction is important
5. Multi-sensor Fusion: How camera data complements LiDAR and IMU

This completes the three-sensor setup demonstrating IMU, LiDAR, and Camera data collection and analysis!

1.4 Multi-Sensor Integration

Sensor Fusion Principles:

- Complementary Strengths: Each sensor type has unique advantages
- Redundancy: Multiple sensors providing similar information
- Temporal Alignment: Synchronizing data from different sensors
- Spatial Registration: Understanding sensor positions relative to each other

Practical Integration:

- Coordinate Frames: Understanding TF relationships between sensors
- Data Correlation: Relating IMU motion to LiDAR scans and camera images
- System Architecture: ROS2 topic structure for multi-sensor systems
- Performance Considerations: Processing requirements and bandwidth

Section 2: Sensor Calibration & Error Correction

Introduction

we address a critical aspect of robotics: sensor calibration. No sensor is perfect - they all have biases, noise, and systematic errors. Calibration is the process of measuring these errors and creating correction parameters to improve sensor accuracy. Proper calibration is essential for reliable robot navigation and perception.

Learning Objectives

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

- Understand the types of errors in different sensors
- Perform systematic calibration procedures for IMU, LiDAR, and cameras
- Evaluate calibration quality and understand its impact on performance
- Apply calibration parameters to live sensor data
- Recognize when re-calibration is necessary

2.1 The Importance of Sensor Calibration

Why Calibration Matters:

- Accuracy Improvement: Correct systematic errors for precise measurements
- Reliability Enhancement: Consistent performance across different conditions
- Sensor Fusion Enablement: Properly aligned data for multi-sensor algorithms
- Long-term Stability: Maintaining performance over time and conditions

Common Sensor Errors:

IMU Errors:

- Bias Errors: Constant offset in measurements
- Scale Factor Errors: Incorrect scaling of sensor readings
- Non-orthogonality: Sensor axes not perfectly aligned
- Temperature Drift: Performance changes with temperature
- Noise: Random variations in measurements

LiDAR Errors:

- Range Bias: Consistent distance measurement errors
- Angular Misalignment: Incorrect angle measurements
- Beam Offset: Laser not centered in housing
- Time Synchronization: Scan timing mismatches with robot motion

Camera Errors:

- Radial Distortion: Straight lines appear curved
- Tangential Distortion: Lens not parallel to sensor
- Focal Length Errors: Incorrect distance perception
- Principal Point Offset: Optical center miscalibration

2.2 IMU Calibration Methodology

Calibration Procedure:

1. Stationary Phase:
   - Keep robot completely still
   - Measure gyroscope bias (should be zero) (0 , 0, 9.8)
   - Measure accelerometer bias relative to gravity
2. Multi-position Phase:
   - Place robot in different orientations
   - Measure gravity vector in each position
   - Calculate accelerometer scale factors
3. Rotation Phase:
   - Rotate robot around each axis
   - Measure gyroscope scale factors
   - Verify angular rate measurements

Key Parameters Calculated:

- Gyroscope Bias: [X, Y, Z] offset in rad/s
- Accelerometer Bias: [X, Y, Z] offset in m/s²
- Scale Factors: Correction multipliers for each axis
- Cross-axis Sensitivity: Alignment corrections

Impact of Calibration:

- Before: Orientation drift, incorrect acceleration
- After: Stable orientation, accurate motion detection
- Navigation Improvement: 60-80% better position estimation

Files Implemented:

- `imu_calibrator.py` - Automated IMU calibration procedure
- `imu_calibration_applier.py` - Real-time calibration application
- `imu_calibration.json` - Calibration parameters storage

2.3 LiDAR Calibration Methodology

Calibration Procedure:

1. Wall-based Calibration:
   - Place robot facing flat wall at known distance
   - Measure range bias from expected distance
   - Detect angular offset from wall straightness
2. Pattern-based Calibration:
   - Use known geometric patterns
   - Measure systematic errors in scan geometry
   - Calculate correction parameters
3. Temporal Calibration:
   - Synchronize LiDAR scans with robot motion
   - Compensate for motion during scanning
   - Adjust for mechanical latency

Key Parameters Calculated:

- Range Bias: Constant distance correction
- Angular Offset: Scan angle correction
- Time Compensation: Motion synchronization
- Beam Parameters: Laser characteristics

Impact of Calibration:

- Before: Curved walls, inaccurate object positions
- After: Straight walls, precise obstacle locations
- Mapping Improvement: Clean, accurate environment maps

Files Implemented:

- `lidar_calibrator.py` - Wall-based LiDAR calibration
- `lidar_calibration.json` - Calibration parameters storage

2.4 Camera Calibration Methodology

Calibration Procedure:

1. Chessboard Pattern:
    - Use known geometric pattern (chessboard)
    - Capture images from multiple viewpoints
    - Detect pattern corners with sub-pixel accuracy
2. Intrinsic Calibration:
    - Calculate camera matrix (focal length, optical center)
    - Determine distortion coefficients
    - Compute reprojection error
3. Extrinsic Calibration:
    - Relate camera to other sensors (LiDAR, IMU)
    - Establish coordinate transformations
    - Verify alignment accuracy

Key Parameters Calculated:

- Camera Matrix: Focal lengths and optical center
- Distortion Coefficients: Radial and tangential distortion
- Reprojection Error: Calibration quality metric
- Resolution Parameters: Image dimensions and aspect ratio

Impact of Calibration:

- Before: Distorted images, curved lines
- After: Rectified images, straight lines
- Vision Improvement: Accurate computer vision algorithms

Files Implemented:

- `camera_calibrator.py` - Chessboard-based camera calibration
- `camera_calibration.json` - Calibration parameters storage

2.5 Calibration Quality Assessment

Evaluation Metrics:

- IMU: Bias stability, noise levels, drift rates
- LiDAR: Range accuracy, angular precision, scan consistency
- Camera: Reprojection error, line straightness, feature accuracy

Validation Procedures:

- Cross-validation: Using different data for testing
- Real-world Testing: Performance in actual environments
- Long-term Monitoring: Tracking calibration stability over time

When to Recalibrate:

- Environmental Changes: Temperature, humidity variations
- Physical Impacts: Drops, vibrations, mechanical stress
- Performance Degradation: Noticeable accuracy reduction
- Periodic Maintenance: Regular schedule-based recalibration

2.6 Practical Calibration Workflow

Step-by-Step Process:

1. Preparation:
   - Ensure stable environment
   - Gather calibration targets
   - Verify sensor connectivity
2. Data Collection:
   - Follow systematic procedures
   - Collect sufficient samples
   - Monitor data quality
3. Parameter Calculation:
   - Process calibration data
   - Compute correction parameters
   - Validate results
4. Application:
   - Apply calibration to live data
   - Verify improvement
   - Document parameters

Best Practices:

- Consistent Conditions: Calibrate in similar conditions to operation
- Adequate Sampling: Collect enough data for statistical significance
- Quality Control: Monitor for outliers and errors
- Documentation: Record calibration dates and parameters

IMU Calibration Implementation

File 1: IMU Calibration Node

demo_sensor/imu_calibrator.py

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

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Imu
from std_msgs.msg import String
import numpy as np
import json
import math

class IMUCalibrator(Node):
    def __init__(self):
        super().__init__('imu_calibrator')

        # Subscribe to IMU data
        self.subscription = self.create_subscription(
            Imu,
            '/imu/data',
            self.imu_callback,
            10)

        # Publisher for calibration status
        self.status_pub = self.create_publisher(String,
'/calibration/status', 10)

        # Calibration data storage
        self.gyro_data = []
        self.accel_data = []
        self.calibration_step = 0  # 0: waiting, 1: stationary, 2: rotating,
3: complete
        self.samples_collected = 0
        self.required_samples = 500  # 5 seconds at 100Hz

        # Calibration results
        self.gyro_bias = np.zeros(3)
        self.accel_bias = np.zeros(3)
        self.gyro_scale = np.ones(3)
        self.accel_scale = np.ones(3)

        self.get_logger().info('=== IMU CALIBRATION STARTED ===')
        self.get_logger().info('Step 1: Keep robot COMPLETELY STATIONARY
for 5 seconds')
        self.get_logger().info('This will measure gyro bias and
accelerometer gravity vector')

        # Start calibration process
        self.start_stationary_calibration()

    def start_stationary_calibration(self):
        """Start stationary calibration phase"""
        self.calibration_step = 1
        self.gyro_data = []
        self.accel_data = []
        self.samples_collected = 0

        status_msg = String()
        status_msg.data = "IMU_CALIBRATION_STATIONARY: Keep robot
COMPLETELY STILL for 5 seconds"
        self.status_pub.publish(status_msg)

        self.get_logger().info('🔴 STATIONARY CALIBRATION: Keep robot
STILL!')

    def imu_callback(self, msg):
        """Process IMU data for calibration"""
        if self.calibration_step == 0:
            return

        # Collect data samples
        self.gyro_data.append([msg.angular_velocity.x,
msg.angular_velocity.y, msg.angular_velocity.z])
        self.accel_data.append([msg.linear_acceleration.x,
msg.linear_acceleration.y, msg.linear_acceleration.z])
        self.samples_collected += 1

        # Update progress every 100 samples
        if self.samples_collected % 100 == 0:
            progress = (self.samples_collected / self.required_samples) *
100
            self.get_logger().info(f'Calibration progress:
{progress:.1f}%')
```

```
        # Check if we have enough samples
        if self.samples_collected >= self.required_samples:
            if self.calibration_step == 1:
                self.process_stationary_calibration()
            elif self.calibration_step == 2:
                self.process_rotation_calibration()

    def process_stationary_calibration(self):
        """Process stationary calibration data"""
        self.get_logger().info('📊 Processing stationary calibration
data...')

        # Convert to numpy arrays
        gyro_array = np.array(self.gyro_data)
        accel_array = np.array(self.accel_data)

        # Calculate gyroscope bias (should be zero when stationary)
        self.gyro_bias = np.mean(gyro_array, axis=0)

        # Calculate accelerometer bias (remove gravity from Z-axis)
        accel_mean = np.mean(accel_array, axis=0)
        gravity_magnitude = np.linalg.norm(accel_mean)
        self.accel_bias = accel_mean - np.array([0, 0, gravity_magnitude])

        # Calculate accelerometer scale factors
        expected_gravity = 9.81
        self.accel_scale = expected_gravity / gravity_magnitude *
np.ones(3)

        self.print_calibration_results("STATIONARY")

        # Move to next calibration step
        self.start_rotation_calibration()

    def start_rotation_calibration(self):
        """Start rotation calibration phase"""
        self.calibration_step = 2
        self.gyro_data = []
        self.accel_data = []
        self.samples_collected = 0

        status_msg = String()
        status_msg.data = "IMU_CALIBRATION_ROTATION: Slowly rotate robot
around all axes"
        self.status_pub.publish(status_msg)
        self.get_logger().info('🟡 ROTATION CALIBRATION: Slowly rotate
robot around all axes')
        self.get_logger().info('Rotate around X, Y, and Z axes to measure
scale factors')

    def process_rotation_calibration(self):
        """Process rotation calibration data"""
        self.get_logger().info('📊 Processing rotation calibration
data...')

        # For simplicity, we'll use a basic scale factor estimation
        # In practice, this would involve more sophisticated methods

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

        # Estimate scale factors based on variance (simplified)
        # Real implementation would use known rotation rates
        max_expected_std = 1.0  # rad/s
        self.gyro_scale = max_expected_std / np.maximum(gyro_std, 0.1)

        self.calibration_step = 3
        self.finalize_calibration()

    def finalize_calibration(self):
        """Finalize calibration and save results"""
        self.get_logger().info('✅ IMU CALIBRATION COMPLETE!')

        # Save calibration parameters
        calibration_data = {
            'gyro_bias': self.gyro_bias.tolist(),
            'accel_bias': self.accel_bias.tolist(),
            'gyro_scale': self.gyro_scale.tolist(),
            'accel_scale': self.accel_scale.tolist(),
            'timestamp': self.get_clock().now().nanoseconds
        }

        # Save to file
        with open('imu_calibration.json', 'w') as f:
            json.dump(calibration_data, f, indent=2)

        self.print_final_results()

        status_msg = String()
        status_msg.data = "IMU_CALIBRATION_COMPLETE: Calibration parameters
saved to imu_calibration.json"
```

```python
        self.status_pub.publish(status_msg)

    def print_calibration_results(self, phase):
        """Print calibration results"""
        print("\n" + "="*70)
        print(f"IMU CALIBRATION RESULTS - {phase}")
        print("="*70)
        print("GYROSCOPE BIAS (rad/s):")
        print(f"  X: {self.gyro_bias[0]:.6f} | Y: {self.gyro_bias[1]:.6f} | Z: {self.gyro_bias[2]:.6f}")
        print("  ⚠️  These biases cause orientation drift over time!")

        print("\nACCELEROMETER BIAS (m/s²):")
        print(f"  X: {self.accel_bias[0]:.6f} | Y: {self.accel_bias[1]:.6f} | Z: {self.accel_bias[2]:.6f}")
        print("  ⚠️  These biases cause incorrect gravity measurement!")

        print("\nACCELEROMETER SCALE:")
        print(f"  X: {self.accel_scale[0]:.6f} | Y: {self.accel_scale[1]:.6f} | Z: {self.accel_scale[2]:.6f}")
        print("="*70)

    def print_final_results(self):
        """Print final calibration summary"""
        print("\n" + "="*70)
        print("🎉 IMU CALIBRATION COMPLETE - SUMMARY")
        print("="*70)
        print("CALIBRATION PARAMETERS FOUND:")
        print(f"Gyro Bias:  [{self.gyro_bias[0]:.6f}, {self.gyro_bias[1]:.6f}, {self.gyro_bias[2]:.6f}] rad/s")
        print(f"Accel Bias: [{self.accel_bias[0]:.6f}, {self.accel_bias[1]:.6f}, {self.accel_bias[2]:.6f}] m/s²")
        print(f"Gyro Scale: [{self.gyro_scale[0]:.6f}, {self.gyro_scale[1]:.6f}, {self.gyro_scale[2]:.6f}]")
        print(f"Accel Scale: [{self.accel_scale[0]:.6f}, {self.accel_scale[1]:.6f}, {self.accel_scale[2]:.6f}]")

        print("\nIMPACT OF CALIBRATION:")
        print("✅ Orientation will no longer drift when stationary")
        print("✅ Gravity measurement will be accurate")
        print("✅ Position estimation will be more precise")
        print("✅ Navigation performance will improve significantly")

        print("\nCALIBRATION FILE: imu_calibration.json")
        print("="*70)
```

```
def main(args=None):
    rclpy.init(args=args)
    imu_calibrator = IMUCalibrator()

    try:
        rclpy.spin(imu_calibrator)
    except KeyboardInterrupt:
        imu_calibrator.get_logger().info('IMU calibration interrupted')
    finally:
        imu_calibrator.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
```

```
    main()
```

File 2: IMU Calibration Applier

demo_sensor/imu_calibration_applier.py

```
python
```

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

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Imu
import json
import numpy as np

class IMUCalibrationApplier(Node):
    def __init__(self):
        super().__init__('imu_calibration_applier')

        # Load calibration parameters
        try:
            with open('imu_calibration.json', 'r') as f:
                self.calibration_data = json.load(f)
            self.get_logger().info('✅ IMU calibration parameters loaded
successfully')
        except FileNotFoundError:
            self.get_logger().error('❌ No calibration file found. Run
imu_calibrator first!')
            self.calibration_data = None
            return
        # Subscribe to raw IMU data
        self.subscription = self.create_subscription(
            Imu,
            '/imu/data',
            self.imu_callback,
            10)

        # Publisher for calibrated IMU data
        self.calibrated_pub = self.create_publisher(Imu,
'/imu/data_calibrated', 10)

        self.get_logger().info('🔄 Applying IMU calibration to live
data...')

    def imu_callback(self, msg):
        """Apply calibration to IMU data"""
        if self.calibration_data is None:
            return

        # Create calibrated message
        calibrated_msg = Imu()
        calibrated_msg.header = msg.header
        calibrated_msg.header.frame_id = "imu_link_calibrated"

        # Extract calibration parameters
        gyro_bias = np.array(self.calibration_data['gyro_bias'])
        accel_bias = np.array(self.calibration_data['accel_bias'])
        gyro_scale = np.array(self.calibration_data['gyro_scale'])
        accel_scale = np.array(self.calibration_data['accel_scale'])

        # Apply gyroscope calibration: calibrated = (raw - bias) * scale
        raw_gyro = np.array([msg.angular_velocity.x,
msg.angular_velocity.y, msg.angular_velocity.z])
        calibrated_gyro = (raw_gyro - gyro_bias) * gyro_scale

        calibrated_msg.angular_velocity.x = calibrated_gyro[0]
        calibrated_msg.angular_velocity.y = calibrated_gyro[1]
        calibrated_msg.angular_velocity.z = calibrated_gyro[2]

        # Apply accelerometer calibration
        raw_accel = np.array([msg.linear_acceleration.x,
msg.linear_acceleration.y, msg.linear_acceleration.z])
        calibrated_accel = (raw_accel - accel_bias) * accel_scale

        calibrated_msg.linear_acceleration.x = calibrated_accel[0]
        calibrated_msg.linear_acceleration.y = calibrated_accel[1]
        calibrated_msg.linear_acceleration.z = calibrated_accel[2]

        # Copy orientation (will be improved by calibrated gyro)
        calibrated_msg.orientation = msg.orientation

        self.calibrated_pub.publish(calibrated_msg)

def main(args=None):
    rclpy.init(args=args)
    applier = IMUCalibrationApplier()

    try:
        rclpy.spin(applier)
    except KeyboardInterrupt:
        applier.get_logger().info('IMU calibration applier stopped')
    finally:
        applier.destroy_node()
        rclpy.shutdown()

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

Part 3: LiDAR Calibration Implementation

File 3: LiDAR Calibration Node

demo_sensor/lidar_calibrator.py

python

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

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan
from std_msgs.msg import String
import numpy as np
import json
import math

class LiDARCalibrator(Node):
    def __init__(self):
        super().__init__('lidar_calibrator')

        # Subscribe to LiDAR data
        self.subscription = self.create_subscription(
            LaserScan,
            '/scan',
            self.lidar_callback,
            10)

        # Publisher for calibration status
        self.status_pub = self.create_publisher(String,
'/calibration/status', 10)

        # Calibration data
        self.scan_data = []
        self.calibration_step = 0
        self.required_scans = 50

        # Calibration results
        self.range_bias = 0.0
        self.angular_offset = 0.0
        self.time_compensation = 0.0

        self.get_logger().info('=== LiDAR CALIBRATION STARTED ===')
        self.get_logger().info('Place robot in front of a flat wall for
calibration')
        self.start_wall_calibration()

    def start_wall_calibration(self):
        """Start wall-based calibration"""
        self.calibration_step = 1
        self.scan_data = []

        status_msg = String()
        status_msg.data = "LIDAR_CALIBRATION_WALL: Place robot facing a
flat wall 1-2 meters away"
        self.status_pub.publish(status_msg)

        self.get_logger().info('🔴 WALL CALIBRATION: Place robot facing a
flat wall')
        self.get_logger().info('The wall should be 1-2 meters away for best
results')

    def lidar_callback(self, msg):
        """Process LiDAR scans for calibration"""
        if self.calibration_step == 0:
            return

        self.scan_data.append(msg)

        if len(self.scan_data) % 10 == 0:
            progress = (len(self.scan_data) / self.required_scans) * 100
            self.get_logger().info(f'LiDAR calibration progress:
{progress:.1f}%')

        if len(self.scan_data) >= self.required_scans:
            self.process_wall_calibration()

    def process_wall_calibration(self):
        """Process wall calibration data"""
        self.get_logger().info('📊 Processing wall calibration data...')

        # Analyze multiple scans to find systematic errors
        all_ranges = []
        for scan in self.scan_data:
            ranges = np.array(scan.ranges)
            # Filter invalid readings
            valid_ranges = ranges[(ranges > scan.range_min) & (ranges <
scan.range_max)]
            all_ranges.extend(valid_ranges)

        if not all_ranges:
            self.get_logger().error('❌ No valid LiDAR readings found!
Check wall placement.')
            return

        # Calculate range bias (assuming wall is flat and at known distance)
        # In practice, you'd use a known distance target
        expected_wall_distance = 1.5  # meters
        measured_distance = np.median(all_ranges)
        self.range_bias = measured_distance - expected_wall_distance

        # Detect angular offset by finding the straightest wall section
        best_scan = self.find_best_wall_scan()
        if best_scan is not None:
            self.angular_offset = self.calculate_angular_offset(best_scan)

        self.calibration_step = 2
        self.finalize_lidar_calibration()

    def find_best_wall_scan(self):
        """Find the scan with the clearest wall detection"""
        best_scan = None
        best_score = -1

        for scan in self.scan_data:
            # Score based on number of consistent readings in frontal arc
            frontal_arc = self.get_frontal_arc_readings(scan)
            if len(frontal_arc) > 10:
                consistency = 1.0 / np.std(frontal_arc)  # Higher
consistency = better
                if consistency > best_score:
                    best_score = consistency
                    best_scan = scan

        return best_scan

    def get_frontal_arc_readings(self, scan):
        """Get readings from frontal arc (-30° to +30°)"""
        frontal_indices = []
        for i, angle in enumerate(np.arange(scan.angle_min, scan.angle_max,
scan.angle_increment)):
            angle_deg = math.degrees(angle)
            if -30 <= angle_deg <= 30:  # Frontal arc
                if scan.range_min < scan.ranges[i] < scan.range_max:
                    frontal_indices.append(scan.ranges[i])
        return frontal_indices

    def calculate_angular_offset(self, scan):
        """Calculate angular offset from straight wall"""
        # Find the angle where the wall is straightest
        # This is a simplified implementation
        frontal_readings = []
        frontal_angles = []

        for i, angle in enumerate(np.arange(scan.angle_min, scan.angle_max,
scan.angle_increment)):
            angle_deg = math.degrees(angle)
            if -45 <= angle_deg <= 45:  # Wider frontal arc
                if scan.range_min < scan.ranges[i] < scan.range_max:
                    frontal_readings.append(scan.ranges[i])
                    frontal_angles.append(angle_deg)

        if len(frontal_readings) < 10:
            return 0.0

        # Simple linear fit to find wall angle
        try:
            coeffs = np.polyfit(frontal_angles, frontal_readings, 1)
            slope = coeffs[0]
            # Convert slope to angular offset
            angular_offset = math.degrees(math.atan(slope))
            return angular_offset
        except:
            return 0.0

    def finalize_lidar_calibration(self):
        """Finalize LiDAR calibration"""
        self.get_logger().info('✅ LiDAR CALIBRATION COMPLETE!')

        # Save calibration parameters
        calibration_data = {
            'range_bias': self.range_bias,
            'angular_offset': math.radians(self.angular_offset),
            'time_compensation': self.time_compensation,
            'timestamp': self.get_clock().now().nanoseconds
        }

        with open('lidar_calibration.json', 'w') as f:
            json.dump(calibration_data, f, indent=2)

        self.print_final_results()

        status_msg = String()
        status_msg.data = "LIDAR_CALIBRATION_COMPLETE: Calibration
parameters saved to lidar_calibration.json"
        self.status_pub.publish(status_msg)

    def print_final_results(self):
        """Print final calibration results"""
        print("\n" + "="*70)
        print("🎉 LiDAR CALIBRATION COMPLETE - SUMMARY")
        print("="*70)
        print("CALIBRATION PARAMETERS FOUND:")
        print(f"Range Bias: {self.range_bias:.4f} meters")
        print(f"Angular Offset: {self.angular_offset:.2f} degrees")
        print(f"Time Compensation: {self.time_compensation:.6f} seconds")

        print("\nIMPACT OF CALIBRATION:")
        if abs(self.range_bias) > 0.01:
            print(f"✅ Range measurements corrected by
{self.range_bias:.3f}m")
        if abs(self.angular_offset) > 0.5:
            print(f"✅ Scan angles corrected by
{self.angular_offset:.2f}°")
        print("✅ Walls will appear straighter in maps")
        print("✅ Object positions will be more accurate")
        print("✅ Navigation will be more precise")

        print("\nCALIBRATION FILE: lidar_calibration.json")
        print("="*70)

def main(args=None):
    rclpy.init(args=args)
    lidar_calibrator = LiDARCalibrator()

    try:
        rclpy.spin(lidar_calibrator)
    except KeyboardInterrupt:
        lidar_calibrator.get_logger().info('LiDAR calibration interrupted')
    finally:
        lidar_calibrator.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
```

```
    main()
```

Part 4: Camera Calibration Implementation

File 4: Camera Calibration Node

demo_sensor/camera_calibrator.py

```
python
```

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

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from std_msgs.msg import String
import cv2
from cv_bridge import CvBridge
import numpy as np
import json

class CameraCalibrator(Node):
    def __init__(self):
        super().__init__('camera_calibrator')

        # Subscribe to camera image
        self.subscription = self.create_subscription(
            Image,
            '/camera/image_raw',
            self.image_callback,
            10)

        # Publisher for calibration status
        self.status_pub = self.create_publisher(String,
'/calibration/status', 10)

        self.bridge = CvBridge()
        self.calibration_step = 0
        self.chessboard_size = (7, 6)  # Internal corners
        self.chessboard_square_size = 0.024  # 24mm squares

        # Calibration data
        self.object_points = []  # 3D points in real world space
        self.image_points = []   # 2D points in image plane
        self.calibration_images = []
        self.required_images = 20

        # Calibration results
        self.camera_matrix = None
        self.distortion_coeffs = None

        self.get_logger().info('=== CAMERA CALIBRATION STARTED ===')
        self.get_logger().info(f'Need {self.required_images} images of
chessboard pattern')
        self.get_logger().info('Print a 7x6 chessboard and show it from
different angles')

        self.start_chessboard_calibration()

    def start_chessboard_calibration(self):
        """Start chessboard-based calibration"""
        self.calibration_step = 1

        status_msg = String()
        status_msg.data = "CAMERA_CALIBRATION_CHESSBOARD: Show 7x6
chessboard from different angles and distances"
        self.status_pub.publish(status_msg)

        self.get_logger().info('🔴 CHESSBOARD CALIBRATION: Show chessboard
to camera')
        self.get_logger().info('Move chessboard to different positions and
angles')

    def image_callback(self, msg):
        """Process camera images for calibration"""
        if self.calibration_step == 0:
            return

        try:
            cv_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")
            self.process_calibration_image(cv_image)

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

    def process_calibration_image(self, cv_image):
        """Process image for chessboard detection"""
        gray = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)

        # Find chessboard corners
        ret, corners = cv2.findChessboardCorners(gray,
self.chessboard_size, None)

        if ret:
            # Refine corner positions
            criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,
30, 0.001)
            corners_refined = cv2.cornerSubPix(gray, corners, (11, 11),
(-1, -1), criteria)

            # Create object points for this image
            objp = np.zeros((self.chessboard_size[0] *
self.chessboard_size[1], 3), np.float32)
            objp[:, :2] = np.mgrid[0:self.chessboard_size[0],
0:self.chessboard_size[1]].T.reshape(-1, 2)
            objp *= self.chessboard_square_size

            self.object_points.append(objp)
            self.image_points.append(corners_refined)
            self.calibration_images.append(cv_image.copy())
            # Draw and display the corners
            cv_image_with_corners = cv_image.copy()
            cv2.drawChessboardCorners(cv_image_with_corners,
self.chessboard_size, corners_refined, ret)

            # Show progress
            progress = (len(self.image_points) / self.required_images) *
100
            self.get_logger().info(f'Chessboard detected! Progress:
{progress:.1f}% ({len(self.image_points)}/{self.required_images})')

            # Display image with corners (optional)
            cv2.imshow('Camera Calibration - Chessboard Detected',
cv_image_with_corners)
            cv2.waitKey(500)  # Show for 500ms

            if len(self.image_points) >= self.required_images:
                self.perform_camera_calibration()
        else:
            self.get_logger().info('Chessboard not found in current image -
try different position/angle')

    def perform_camera_calibration(self):
        """Perform camera calibration using collected points"""
        self.get_logger().info('📊 Performing camera calibration...')

        image_size = (self.calibration_images[0].shape[1],
self.calibration_images[0].shape[0])

        # Perform camera calibration
        ret, camera_matrix, distortion_coeffs, rvecs, tvecs =
cv2.calibrateCamera(
            self.object_points, self.image_points, image_size, None, None)

        if ret:
            self.camera_matrix = camera_matrix
            self.distortion_coeffs = distortion_coeffs
            self.calibration_step = 2
            self.finalize_camera_calibration(ret)
        else:
            self.get_logger().error('❌ Camera calibration failed!')

    def finalize_camera_calibration(self, calibration_error):
        """Finalize camera calibration"""
        self.get_logger().info('✅ CAMERA CALIBRATION COMPLETE!')
        # Save calibration parameters
        calibration_data = {
            'camera_matrix': self.camera_matrix.tolist(),
            'distortion_coefficients': self.distortion_coeffs.tolist(),
            'calibration_error': float(calibration_error),
            'image_width': self.calibration_images[0].shape[1],
            'image_height': self.calibration_images[0].shape[0],
            'timestamp': self.get_clock().now().nanoseconds
        }

        with open('camera_calibration.json', 'w') as f:
            json.dump(calibration_data, f, indent=2)

        self.print_final_results(calibration_error)

        status_msg = String()
        status_msg.data = "CAMERA_CALIBRATION_COMPLETE: Calibration parameters saved to camera_calibration.json"
        self.status_pub.publish(status_msg)

        # Close any open windows
        cv2.destroyAllWindows()

    def print_final_results(self, calibration_error):
        """Print final calibration results"""
        print("\n" + "="*70)
        print("🎉 CAMERA CALIBRATION COMPLETE - SUMMARY")
        print("="*70)
        print("CALIBRATION PARAMETERS FOUND:")
        print(f"Calibration Error: {calibration_error:.4f} pixels")
        print("\nCAMERA MATRIX (Intrinsic Parameters):")
        print(f"Focal Length: [{self.camera_matrix[0,0]:.2f}, {self.camera_matrix[1,1]:.2f}]")
        print(f"Optical Center: [{self.camera_matrix[0,2]:.2f}, {self.camera_matrix[1,2]:.2f}]")

        print("\nDISTORTION COEFFICIENTS:")
        print(f"Radial: [{self.distortion_coeffs[0,0]:.4f}, {self.distortion_coeffs[0,1]:.4f}, {self.distortion_coeffs[0,4]:.4f}]")
        print(f"Tangential: [{self.distortion_coeffs[0,2]:.4f}, {self.distortion_coeffs[0,3]:.4f}]")

        print("\nIMPACT OF CALIBRATION:")
        print("✅ Straight lines will appear straight in images")
        print("✅ Distance measurements will be accurate")
        print("✅ Computer vision algorithms will work correctly")
        print("✅ 3D reconstruction will be precise")

        print("\nCALIBRATION FILE: camera_calibration.json")
        print("="*70)

def main(args=None):
    rclpy.init(args=args)
    camera_calibrator = CameraCalibrator()

    try:
        rclpy.spin(camera_calibrator)
    except KeyboardInterrupt:
        camera_calibrator.get_logger().info('Camera calibration
interrupted')
        cv2.destroyAllWindows()
    finally:
        camera_calibrator.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
```

```
    main()
```

Part 5: Updated Setup and Launch Files

Updated setup.py

```python
    entry_points={
        'console_scripts': [
            'imu_display = demo_sensor.imu_display:main',
            'robot_mover = demo_sensor.robot_mover:main',
            'lidar_display = demo_sensor.lidar_display:main',
            'lidar_visualizer = demo_sensor.lidar_visualizer:main',
            'camera_display = demo_sensor.camera_display:main',
            'camera_visualizer = demo_sensor.camera_visualizer:main',
            # Calibration nodes
            'imu_calibrator = demo_sensor.imu_calibrator:main',
            'imu_calibration_applier =
demo_sensor.imu_calibration_applier:main',
            'lidar_calibrator = demo_sensor.lidar_calibrator:main',
            'camera_calibrator = demo_sensor.camera_calibrator:main',
        ],
    },
```

Calibration Launch File

launch/calibration.launch.py

```python
from launch import LaunchDescription
from launch.actions import ExecuteProcess, TimerAction
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory
import os

def generate_launch_description():
    package_name = 'demo_sensor'
    package_share = get_package_share_directory(package_name)

    # Robot URDF file
    urdf_file = os.path.join(package_share, 'urdf',
'simple_robot.urdf.xacro')

    # World file with calibration features
    world_file = os.path.join(package_share, 'worlds', 'sensor_test.world')

    # Basic nodes
    robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        arguments=[urdf_file]
    )

    spawn_entity = Node(
        package='gazebo_ros',
        executable='spawn_entity.py',
        arguments=['-entity', 'simple_robot', '-topic',
'robot_description', '-x', '0.0', '-y', '0.0', '-z', '0.1'],
        output='screen'
    )
    gazebo = ExecuteProcess(
        cmd=['gazebo', '--verbose', world_file, '-s',
'libgazebo_ros_init.so',
             '-s', 'libgazebo_ros_factory.so'],
        output='screen'
    )

    # Calibration nodes (start with delay)
    imu_calibrator = Node(
        package=package_name,
        executable='imu_calibrator',
        name='imu_calibrator',
        output='screen'
    )

    lidar_calibrator = Node(
        package=package_name,
        executable='lidar_calibrator',
        name='lidar_calibrator',
        output='screen'
    )

    camera_calibrator = Node(
        package=package_name,
        executable='camera_calibrator',
        name='camera_calibrator',
        output='screen'
    )

    # Start calibration nodes after 5 seconds
    delayed_calibration = TimerAction(
        period=5.0,
        actions=[imu_calibrator, lidar_calibrator, camera_calibrator]
    )

    return LaunchDescription([
        gazebo,
        robot_state_publisher,
        spawn_entity,
        delayed_calibration,

    ])
```

Usage Instructions

1. Build and Run Calibration

```
bash
```

```
cd ~/ros2_ws
colcon build --packages-select demo_sensor
source install/setup.bash

# Run calibration session
```

```
ros2 launch demo_sensor calibration.launch.py
```

2. Follow Calibration Steps

IMU Calibration:

- Keep robot COMPLETELY STILL for 5 seconds
- Then slowly rotate robot around all axes

LiDAR Calibration:

- Place robot facing a flat wall 1-2 meters away
- Keep robot stationary during calibration

Camera Calibration:

- Print a 7x6 chessboard pattern
- Show it to camera from different angles and distances
- Need 20 good detections

3. Apply Calibration

```
bash
```

```
# Apply IMU calibration to live data
```

```
ros2 run demo_sensor imu_calibration_applier
```

Conclusion

section 1 Recap:

You learned how to integrate three essential robot sensors (IMU, LiDAR, Camera) into a ROS2 system, visualize their data in real-time, and understand the fundamental principles of each sensor type. This foundation is crucial for effective robot perception and navigation.

section 2 Recap:

You discovered why sensor calibration is essential and learned systematic procedures to calibrate each sensor type. Proper calibration transforms raw, inaccurate sensor data into precise, reliable measurements that enable robust robot operation.

Key Takeaways:

1. Sensors are Imperfect: All sensors have inherent errors that must be corrected
2. Calibration is Essential: Proper calibration dramatically improves robot performance
3. Systematic Approach: Follow structured procedures for reliable results
4. Continuous Process: Calibration should be part of regular robot maintenance