University Rover Challenge

Complete Technical Documentation

& Startup Guide

University of Pittsburgh Robotics Club

ROS 2 Humble • NVIDIA Jetson Nano • Docker

Version 2.0 — January 2026

Platform:	ROS 2 Humble Hawksbill
Hardware:	NVIDIA Jetson Nano, Arduino
Deployment:	Docker Containers
GUI:	PyQt5

Contents

List of Figures

List of Tables

Chapter 1

Project Overview

1.1 Introduction

This document provides comprehensive technical documentation for the University of Pittsburgh Robotics Club's University Rover Challenge (URC) rover system. The system integrates advanced robotics technologies including:

- **6-wheel differential drive system** with GoBilda motors
- **Multi-sensor fusion** including GPS, IMU, ultrasonic sensors, and cameras
- **ROS 2 Humble** for distributed robot communication
- **Docker containerization** for reproducible deployments
- **PyQt5 GUI system** for teleoperation and monitoring

1.2 Competition Tasks

The University Rover Challenge consists of multiple mission types:

Table 1.1: URC Competition Tasks

Task	**Description**
Autonomous Navigation	GPS-guided traversal of challenging Martian-analog terrain
Equipment Servicing	Precise manipulation tasks using robotic arm
Extreme Delivery	Package delivery in harsh environmental conditions
Science Operations	Sample collection and in-field analysis
Teleoperation	Manual control for complex scenarios

1.3 Key Specifications

Table 1.2: Rover System Specifications

Metric	Value
Navigation Accuracy	±2m GPS positioning
Video Latency	<200ms streaming
Control Response	<50ms command execution
Battery Life	4+ hours autonomous operation
Operating Range	1km+ radio communication
Motor Control	6 GoBilda motors (0–255 PWM)
Camera Frame Rate	30 Hz
IMU Update Rate	10 Hz
Serial Baud Rate	115200 bps

1.4 How Systems Work Together

The URC rover system follows a distributed architecture where components work together through a multi-layered communication stack. Understanding this flow is essential for developers extending or debugging the system.

1.4.1 End-to-End Data Flow

When an operator interacts with the system, data flows through the following path:

1. **User Input** — Operator presses a key or moves a joystick on the base station
2. **GUI Publisher** — PyQt5 GUI captures input and calls `MotorPublisher.publish_motor_command()` or `TwistPublisher.publish_twist()`
3. **ROS 2 Topic** — Publisher sends message to topic (e.g., `/motor_control_input` or `/cmd_vel`) over WiFi via ROS 2 DDS
4. **Bridge Subscriber** — `MotorBridge` on Jetson receives message via subscription callback
5. **Serial Transmission** — Bridge formats command as comma-separated string and writes to Arduino via `serial.write()` at 115200 baud
6. **Arduino Processing** — Arduino firmware parses command, calculates differential drive values
7. **Motor Control** — Arduino sets PWM and direction pins to drive GoBilda motors

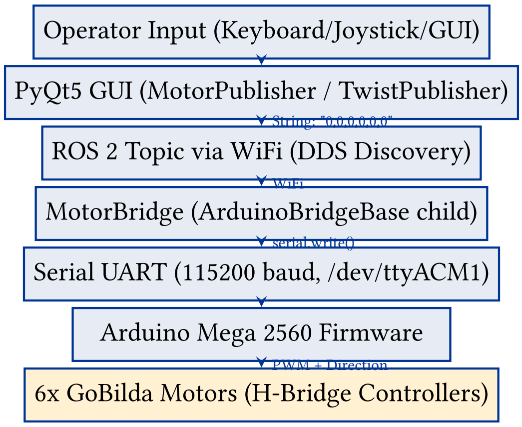

Figure 1.1: Complete Motor Control Data Flow — Note: `/dev/ttyACM1` assignment requires Motor Arduino to be plugged in **second**. See Section 13.2.3 for required plug-in order.

Important

Hardware Requirement: Two Arduino Boards
This system requires **two separate Arduino boards**:

1. **Sensor Arduino** (plugged in first → `/dev/ttyACM0`) — Runs IMU, GPS, and ultrasonic sensor firmware
2. **Motor Arduino** (plugged in second → `/dev/ttyACM1`) — Runs motor control firmware for 6-wheel drive

The port numbers are assigned based on USB plug-in order, not physical port location. See Section 13.2.3 for detailed plug-in procedures.

1.4.2 Sensor Data Return Path

Sensor data flows in the reverse direction:

1. **Hardware Sensor** — BNO055 IMU reads orientation, GPS receives NMEA sentences
2. **Arduino** — Formats sensor data and sends via `Serial.println()`
3. **Bridge Timer** — `ArduinoBridgeBase.read_from_arduino()` executes every 100ms via ROS 2 timer
4. **ROS 2 Publisher** — Bridge publishes data to topic (e.g., `/imu_data`, `/gps_data`)
5. **GUI Subscriber** — `IMUSubscriber` or `GPSSubscriber` receives message, parses with data parser
6. **PyQt5 Signal** — Subscriber emits `imu_data_updated` signal with parsed values
7. **GUI Update** — Signal triggers slot function that updates UI labels/displays

1.4.3 Base Station vs. Rover Responsibilities

Table 1.3: Component Location and Purpose

Component	Location	Responsibility
GUI Applications	Base Station	User interface, input capture, data visualization
Publisher Classes	Base Station	Convert user actions to ROS 2 messages
Subscriber Classes	Base Station	Receive sensor data, parse, update GUI
Data Parsers	Base Station	Parse raw sensor strings into usable values
Bridge Classes	Rover (Jetson)	ROS 2 ↔ Serial translation
Supervisor	Rover (Jetson)	Process management, auto-restart
Arduino Firmware	Rover (Arduino)	Hardware control, sensor reading
Motors/Sensors	Rover (Physical)	Physical actuation and sensing

Tip

When to modify each component:

- **Add new GUI control** → Modify/create Publisher class in `guis/publishers/`
- **Add new sensor display** → Create Subscriber class in `guis/subscribers/` with data parser
- **Change serial protocol** → Modify Bridge class in `ros_bridge/` and corresponding Arduino sketch
- **Add new hardware** → Create new Bridge class inheriting from `ArduinoBridgeBase`

Chapter 2

System Architecture

2.1 High-Level Architecture

The rover system uses a **dual-container architecture** for optimal performance and separation of concerns.

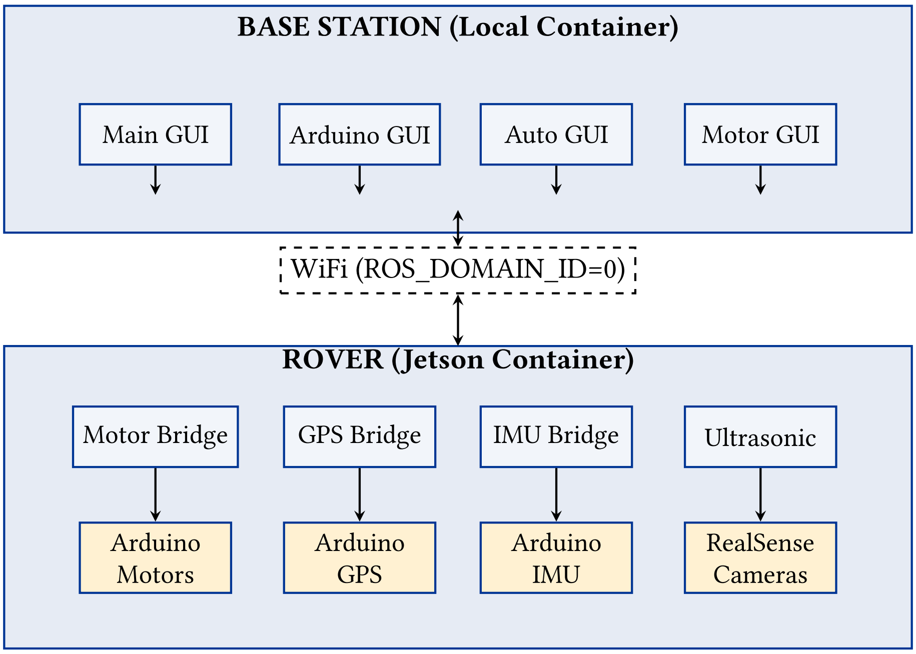

Figure 2.1: System Architecture Overview

2.2 Dual-Container Architecture

Table 2.1: Container Responsibilities

Container	Location	Purpose
Jetson Container	On-board rover	Hardware bridges, sensor acquisition, motor control, camera feeds
Local Container	Base station	GUI interfaces, visualization, high-level control, debugging

Tip

The dual-container architecture saves Jetson processing power for real-time control while enabling easier debugging with local GUI applications. ROS 2 communication works seamlessly over WiFi.

2.3 Communication Flow

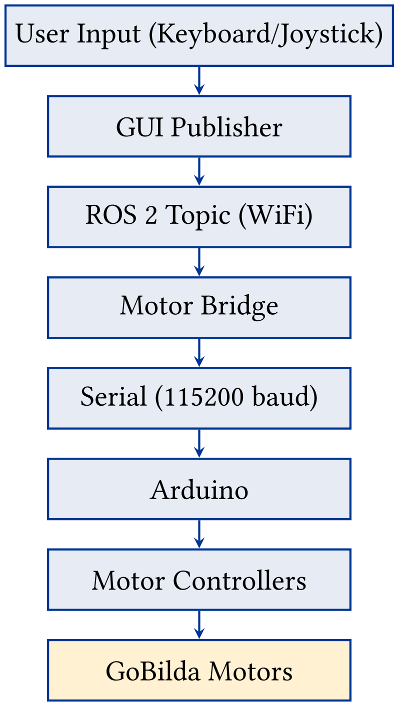

Figure 2.2: Motor Control Communication Flow

2.4 Sensor Data Flow

Each sensor follows a parallel data pipeline from hardware to GUI display. Understanding these pipelines helps developers debug sensor issues and add new sensor types.

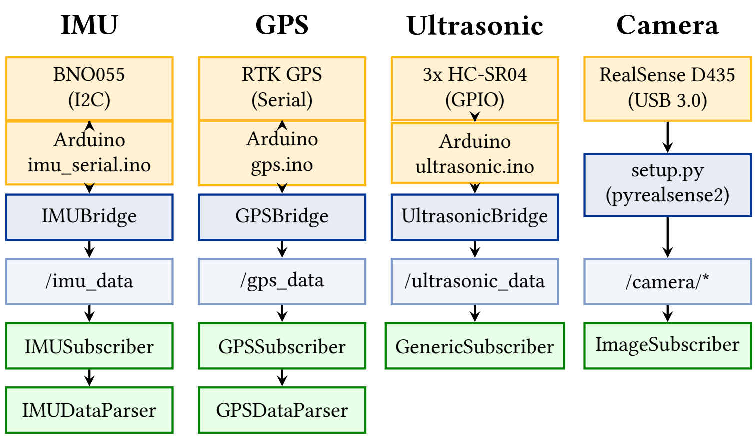

Figure 2.3: Parallel Sensor Data Pipelines

2.5 Error Handling Architecture

The system implements error handling at multiple layers to ensure graceful degradation when components fail.

2.5.1 Serial Connection Errors

The `ArduinoBridgeBase` class handles serial connection failures:

```python
def _attempt_reconnection(self):
    """Attempt to reconnect to the serial port"""
    if self.serial:
        try:
            self.serial.close()
        except:
            pass
        self.serial = None

    # Manual restart required - logs warning
    self.get_logger().warning(
        "Serial connection lost. Manual restart required."
    )
```

Listing 2.1: Reconnection Logic in ArduinoBridgeBase

2.5.2 Data Parsing Errors

Subscribers use sentinel values to indicate parse failures:

Table 2.2: Error Values by Subscriber

Subscriber	Error Value	Condition
IMUSubscriber	-0.1	Parse exception in `IMUDataParser`
GPSSubscriber	"NO CONNECTION!!!"	Empty NMEA sentence received
GPSSubscriber	None	Invalid sentence type (not GAGSV/GNGLL)

> **Warning**
>
> When displaying sensor data in GUIs, always check for error values before showing data to the user. A value of -0.1 for IMU data indicates a parsing failure, not an actual sensor reading.

2.5.3 IMUBridge Port Auto-Detection

The `IMUBridge` class implements automatic port detection for cross-platform compatibility:

```python
possible_ports = [
    "/dev/ttyACM0",      # Linux primary
    "/dev/cu.usbmodem11201",  # macOS
    "/dev/ttyUSB0"       # Linux USB-serial
]

for port in possible_ports:
    try:
        test_serial = serial.Serial(port, 115200, timeout=1)
        test_serial.close()
        working_port = port
```

```
        break
    except:
        continue
```

Listing 2.2: IMU Port Auto-Detection

2.6 Message Format Specifications

Detailed field-by-field breakdown of all message formats used in the system.

2.6.1 Motor Control Input

Table 2.3: Motor Command String Format

Field	Name	Description
0	linear_x	Forward/backward velocity (-100 to 100)
1	linear_y	Left/right strafe (typically 0)
2	linear_z	Up/down (typically 0)
3	angular_x	Roll (typically 0)
4	angular_y	Pitch (typically 0)
5	angular_z	Yaw/turning (-100 to 100)

Example: `"100,0,0,0,0,50"` = Forward at full speed with moderate right turn

2.6.2 IMU Data Format

```
Format: "X: <float>\tY: <float>\tZ: <float>"
Fields: X = heading (0-360), Y = roll, Z = pitch
Example: "X: 45.23\tY: -2.15\tZ: 0.89"
```

Listing 2.3: IMU String Format

The `IMUDataParser` processes this into:

- **distance** — Euclidean distance from previous reading
- **velocity** — distance / time_delta (meters/second)
- **yz_tilt_angle** — Vertical tilt in degrees
- **yx_tilt_angle** — Horizontal tilt in degrees

2.6.3 GPS NMEA Formats

GNGLL (Position):

```
$GNGLL,<lat>,<N/S>,<lon>,<E/W>,<time>,<status>,<mode>*<checksum>
Example: $GNGLL,4024.12345,N,07952.12345,W,123456.00,A,A*6B

Parsed output: "Latitude: 4024.12345 N Longitude: 07952.12345 W"
```

Listing 2.4: GNGLL Sentence Structure

GAGSV/GBGSV (Satellites):

```
$GAGSV,<total>,<msg#>,<sats>,<sat1_prn>,<elev>,<azim>,<snr>,...*<checksum>
SNR values extracted from positions 7, 11, 15, 19... (every 4th field)

Parsed output: List of SNR integers, average SNR float
```

Listing 2.5: GAGSV Sentence Structure

2.6.4 Ultrasonic Data Format

```
Format: "distance1_cm, distance2_cm, distance3_cm, "
Example: "45, 120, 88, "
Note: Trailing comma and space are part of the format
```

Listing 2.6: Ultrasonic String Format

Chapter 3

Directory Structure

3.1 Project Layout

The project follows a modular organization:

```
URC/
|-- docker/                     # Docker configuration
|   |-- jetson/                # On-board Jetson container
|   |   |-- Dockerfile
|   |   |-- docker-compose.yml
|   |   |-- start.sh
|   |   |-- supervisord.conf
|   |   +-- setup.py
|   |-- jetson_local/          # Alternative Jetson setup
|   +-- local/                 # Base station container
|       |-- Dockerfile
|       |-- docker-compose.yml
|       |-- start_mac.sh
|       |-- start_linux.sh
|       +-- sim-launch.sh
|
|-- ros_bridge/                # ROS 2 hardware bridges
|   |-- arduino_bridge_base/   # Base class for bridges
|   |-- motor_bridge/          # Motor control system
|   |-- motor_subscriber/      # Legacy motor control
|   |-- gps_bridge/            # GPS sensor bridge
|   |-- imu_bridge/            # IMU sensor bridge
|   +-- ultrasonic_bridge/     # Ultrasonic sensor bridge
|
|-- guis/                      # PyQt5 GUI applications
|   |-- gen_gui.py             # Main control dashboard
|   |-- arduino_gui.py         # Hardware control interface
|   |-- auto_gui.py            # Autonomous navigation
|   |-- json_motorGUI.py       # Motor/Arm keyboard control
|   |-- publishers/            # ROS 2 publisher classes
|   |-- subscribers/           # ROS 2 subscriber classes
|   +-- camera/                # Camera utilities
|
|-- simulation/                # Gazebo simulation
|   +-- colcon_ws/             # ROS 2 workspace
|
|-- README.md
|-- CONTRIBUTING.md
+-- Makefile
```

Listing 3.1: Project Directory Structure

3.2 Key Directories

3.2.1 Docker Configuration

- `docker/jetson/` — On-board rover container with NVIDIA runtime support
- `docker/local/` — Base station container with X11 forwarding for GUI
- `docker/jetson_local/` — Alternative hybrid setup

3.2.2 ROS Bridge

- `arduino_bridge_base/` — Base class providing serial communication (248 lines)
- `motor_bridge/` — Unified motor control with multiple input sources
- `gps_bridge/` — NMEA sentence parsing and publishing
- `imu_bridge/` — BNO055 9-axis IMU integration
- `ultrasonic_bridge/` — Multi-sensor distance measurement

3.2.3 GUI System

- `gen_gui.py` — Main dashboard (623 lines) with camera feeds and sensor data
- `arduino_gui.py` — Joystick-style motor control (617 lines)
- `json_motorGUI.py` — Keyboard hotkey control (292 lines)
- `publishers/` — MotorPublisher, TwistPublisher classes
- `subscribers/` — IMUSubscriber, GPSSubscriber, data parsers

3.3 Module Interdependencies

Understanding how modules depend on each other helps developers know which files to modify when extending the system.

3.3.1 Class Inheritance Hierarchy

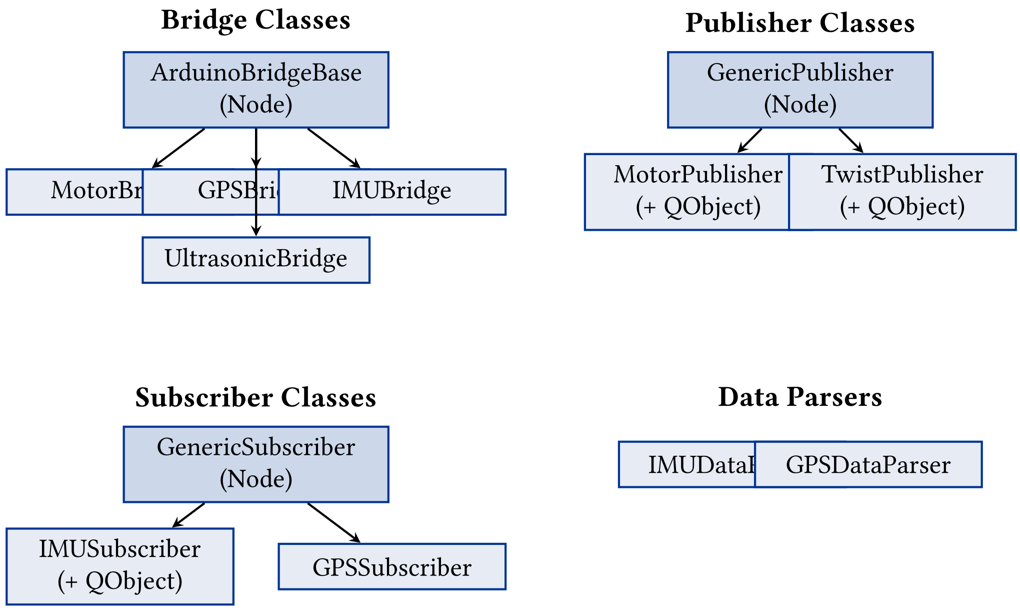

Figure 3.1: Module Inheritance and Composition

3.3.2 Import Dependencies

Table 3.1: Module Import Requirements

Module	**Key Imports**
ArduinoBridgeBase	`rclpy`, `serial`, `std_msgs.msg.String`
MotorBridge	ArduinoBridgeBase, `geometry_msgs.msg.Twist`
GenericPublisher	`rclpy.node.Node`
MotorPublisher	GenericPublisher, `PyQt5.QtCore.QObject`, `pyqtSignal`
GenericSubscriber	`rclpy.node.Node`
IMUSubscriber	GenericSubscriber, IMUDataParser, `PyQt5.QtCore`
GPSSubscriber	GenericSubscriber, GPSDataParser

3.3.3 sys.path Configuration

GUI modules require path configuration to import sibling packages:

```python
import sys
import os

# Add parent directory to path for sibling imports
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

# Now can import from publishers/ and subscribers/
from guis.publishers.publisher import MotorPublisher
from guis.subscribers.subscriber import IMUSubscriber
```

Listing 3.2: Import Path Setup (from gen_gui.py)

Tip

When adding new modules:

- Place publishers in `guis/publishers/` and inherit from `GenericPublisher`
- Place subscribers in `guis/subscribers/` and inherit from `GenericSubscriber`
- Place bridges in `ros_bridge/<bridge_name>/` and inherit from `ArduinoBridgeBase`
- Data parsers are standalone classes in `guis/subscribers/`

Chapter 4

Hardware Requirements

4.1 On-Board Computer

Table 4.1: Jetson Nano Specifications

Component	Specification
Computer	NVIDIA Jetson Nano (4GB)
Storage	64GB+ SD Card or NVMe SSD
Power	5V 4A barrel jack
Cooling	Active fan (recommended)

4.2 Motor System

Table 4.2: Motor System Specifications

Component	Specification
Motors	6x GoBilda motors
Motor Controllers	3x dual H-bridge controllers
Arduino	Arduino Mega 2560 (recommended)
PWM Range	0–255
Wheel Base	0.7384 meters
Max Speed	1.0 m/s

4.2.1 Pin Configuration

Table 4.3: Arduino Motor Pin Configuration

Motor	PWM Pin	Direction Pin	Side
Front Left	2	53	Left
Middle Left	4	51	Left
Back Left	6	49	Left
Front Right	3	52	Right
Middle Right	5	50	Right
Back Right	7	48	Right

4.3 Sensors

Table 4.4: Sensor Configuration

Sensor	Model	Connection	Rate
IMU	Adafruit BNO055 (9-axis)	Serial 115200	10 Hz
GPS	RTK GPS module	Serial 115200	1 Hz
Cameras	2x Intel RealSense D435	USB 3.0	30 Hz
Ultrasonic	3x HC-SR04	Arduino GPIO	10 Hz

4.3.1 Ultrasonic Sensor Pins

Table 4.5: Ultrasonic Sensor Pin Configuration

Sensor	Trigger Pin	Echo Pin
Sensor 1	8	9
Sensor 2	50	51
Sensor 3	48	49

4.4 Serial Port Configuration

Table 4.6: Serial Port Assignments — Port numbers depend on USB plug-in order (see Section 13.2.3)

Device	Port	Baud Rate	Arduino	Notes
Motors	/dev/ttyACM1	115200	Motor (2nd)	Hardcoded in MotorBridge
GPS	/dev/ttyACM0	115200	Sensor (1st)	Configurable, default ACM0
IMU	/dev/ttyACM0	115200	Sensor (1st)	Auto-detect tries ACM0 first
Ultrasonic	/dev/ttyACM0	115200	Sensor (1st)	Same Arduino as IMU/GPS

Tip

Which Arduino? The "Sensor (1st)" Arduino handles all sensor data (IMU, GPS, ultrasonic) while the "Motor (2nd)" Arduino handles motor commands. The 1st/2nd refers to the required USB plug-in order on the Jetson.

Chapter 5

Software Dependencies

5.1 System Requirements

- **Operating System:** Ubuntu 22.04 LTS
- **ROS Version:** ROS 2 Humble Hawksbill
- **Python:** 3.10+
- **Docker:** 20.10+

5.2 Python Dependencies

```
# Core ROS 2
rclpy
ros2cli

# Hardware Communication
pyserial>=3.5
pyrealsense2>=2.50.0

# GUI Framework
PyQt5>=5.15.0
PyQt5-sip

# Computer Vision
opencv-python>=4.5.0
numpy<2.0

# ROS 2 Message Bridges
cv_bridge
sensor_msgs
geometry_msgs
std_msgs
```

Listing 5.1: Python Package Requirements

5.3 System Packages (APT)

```
# Base development
python3-pip python3-pyqt5 python3-pyqt5.qtsvg

# Graphics libraries (for GUI)
```

```
libxcb-xinerama0 libxcb-cursor0 libxkbcommon-x11-0
libgl1-mesa-glx libgl1-mesa-dri mesa-utils x11-apps

# ROS 2 Humble
ros-humble-desktop ros-dev-tools
ros-humble-teleop-twist-keyboard ros-humble-joy

# Intel RealSense
librealsense2-utils librealsense2-dev
ros-humble-librealsense2-* ros-humble-realsense2-*
```

Listing 5.2: System Package Requirements

5.4 Arduino Libraries

- `Adafruit_Sensor` — Unified sensor abstraction layer
- `Adafruit_BNO055` — 9-axis IMU driver library

5.5 Dependency Rationale

Understanding why each dependency is required helps developers troubleshoot installation issues and evaluate alternatives.

Table 5.1: Python Package Rationale

Package	Version	Used By	Why Required
pyserial	≥3.5	ArduinoBridgeBase	Serial communication with Arduino; 3.5+ includes stability fixes for Linux
pyrealsense2	≥2.50.0	setup.py, ImageSubscriber	Intel RealSense SDK bindings; 2.50+ required for D435 depth camera
PyQt5	≥5.15.0	All GUIs	Qt bindings for GUI framework; 5.15+ has stable signal/slot mechanism
opencv-python	≥4.5.0	Camera utilities	Image processing and cv_bridge integration
numpy	<2.0	IMUDataParser, cameras	Matrix operations; 2.0 has breaking API changes affecting cv_bridge

5.5.1 Version Constraint Explanations

Important

numpy<2.0 Constraint
NumPy 2.0 introduced breaking changes to array handling that affect `cv_bridge` and OpenCV integration. The constraint is enforced in both Dockerfiles:

```
# From docker/jetson/Dockerfile and docker/local/Dockerfile
RUN pip3 install 'numpy<2.0' opencv-python
```

If you upgrade to numpy 2.0+, expect `TypeError` exceptions when converting ROS Image messages to OpenCV arrays.

5.5.2 ROS 2 Humble Compatibility

ROS 2 Humble (May 2022 – May 2027) is the current LTS release. Key compatibility notes:

- **Python 3.10+** — Required by Humble; earlier Python versions will fail
- **Ubuntu 22.04** — Official supported platform for Humble
- **DDS Implementation** — Uses Fast-RTPS by default (`RMW_IMPLEMENTATION=rmw_fastrtps_cpp`)

5.5.3 Development vs. Production Dependencies

Table 5.2: Dependency Categories

Category	**Packages**	**Notes**
Core (Required)	rclpy, pyserial, PyQt5	Essential for all operations
Sensors	pyrealsense2, Adafruit libraries	Only needed with corresponding hardware
Development	ros-dev-tools, x11-apps	Debugging and testing tools
Simulation	Gazebo packages	Only for simulation mode

Chapter 6

Docker Configuration

6.1 Jetson Container

6.1.1 Overview

The Jetson container runs on-board the rover and manages all hardware interfaces.

Warning

CRITICAL: Re-Plug Arduinos Before EVERY Docker Start
The Docker container captures serial port mappings at startup. You **MUST** unplug and re-plug the Arduinos in the correct order **every time** before starting the container:

1. Unplug all Arduinos from Jetson
2. Plug in **Sensor Arduino FIRST** → gets `/dev/ttyACM0`
3. Plug in **Motor Arduino SECOND** → gets `/dev/ttyACM1`
4. Verify with `ls /dev/ttyACM*`
5. **Then** start the Docker container

See Section 13.2.3 for details.

```
cd docker/jetson
sudo docker build -t urc_jetson .
# FIRST: Re-plug Arduinos in correct order (Sensor, then Motor)
sudo docker-compose up -d
```

Listing 6.1: Build and Start Jetson Container

6.1.2 Key Features

- NVIDIA GPU support (`runtime: nvidia`)
- Privileged mode for device access
- Host network mode for ROS 2 communication
- Supervisor process management

6.1.3 Device Mappings

```
devices:
  - /dev:/dev                  # All devices
  - /dev/bus/usb:/dev/bus/usb    # USB devices
  - /dev/video0:/dev/video0      # Camera 1
  - /dev/video1:/dev/video1      # Camera 2
```

Listing 6.2: Docker Compose Device Configuration

6.1.4 Environment Variables

```
environment:
  - ROS_DOMAIN_ID=0
  - ROS_LOCALHOST_ONLY=0
  - NVIDIA_DRIVER_CAPABILITIES=all
  - NVIDIA_VISIBLE_DEVICES=all
```

Listing 6.3: Jetson Environment Variables

6.1.5 Supervised Processes

Table 6.1: Supervisor Managed Processes

Process	Description	Auto-restart
setup	RealSense camera initialization	Yes
gps_bridge	GPS data publisher	Yes
motor_bridge	Motor command handler	Yes
ultrasonic_bridge	Ultrasonic sensor publisher	Yes

6.2 Local Container (Base Station)

6.2.1 macOS Setup

```
# Prerequisite: Install XQuartz and enable network clients
# System Preferences > Security > Allow connections

cd docker/local
chmod +x start_mac.sh
./start_mac.sh
```

Listing 6.4: macOS Container Startup

6.2.2 Linux Setup

```
cd docker/local
chmod +x start_linux.sh
./start_linux.sh
```

Listing 6.5: Linux Container Startup

6.2.3 Local Environment Variables

```yaml
environment:
  - ROS_DOMAIN_ID=0
  - ROS_LOCALHOST_ONLY=0
  - QT_X11_NO_MITSHM=1
  - DISPLAY=${DISPLAY}
```

Listing 6.6: Local Container Environment

6.3 Simulation Container

```bash
cd docker/local
./sim-launch.sh

# Launch simulation
ros2 launch my_robot_description display.launch.py use_sim_time:=true
```

Listing 6.7: Simulation Startup

Chapter 7

Quick Start Guide

7.1 Prerequisites

7.1.1 Install Docker

```
# Linux
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh

# macOS
brew install --cask docker
```

Listing 7.1: Docker Installation

7.1.2 Clone Repository

```
git clone https://github.com/pitt-robotics/URC.git
cd URC
```

Listing 7.2: Clone Repository

7.1.3 Connect Hardware

Warning

CRITICAL: Arduino Plug-In Order Determines Port Numbers
The order you plug in the Arduinos determines their `/dev/ttyACM*` port numbers. You must follow this exact order **every time before starting Docker**:

1. **Ensure Arduinos are UNPLUGGED** from Jetson
2. Power on Jetson and wait for full boot (60-90 seconds)
3. **Plug in SENSOR Arduino FIRST** → becomes `/dev/ttyACM0`
   - This Arduino runs IMU, GPS, and ultrasonic firmware
   - Wait 2 seconds after plugging in
4. **Plug in MOTOR Arduino SECOND** → becomes `/dev/ttyACM1`
   - This Arduino runs motor control firmware

   - Wait 2 seconds after plugging in

5. Connect RealSense cameras to USB 3.0 ports
6. Power on motor controllers
7. Verify serial ports: `ls /dev/ttyACM*`
   - Must show both `ttyACM0` and `ttyACM1`
8. **NOW** start the Docker container

Tip

Why This Matters: Docker captures the `/dev/ttyACM*` mappings when the container starts. The MotorBridge is hardcoded to use `/dev/ttyACM1`, so the Motor Arduino **must** be the second device plugged in.

7.2 Option A: Full System Startup

Important

This option requires both the rover (Jetson) and base station (local machine) to be on the same WiFi network.

7.2.1 Step 1: Start Jetson Container (on rover)

Warning

Before running start.sh: Unplug and re-plug Arduinos in correct order (Sensor first → ACM0, Motor second → ACM1). The container captures port assignments at startup!

```
# FIRST: Re-plug Arduinos (Sensor, then Motor)
cd docker/jetson
sudo ./start.sh
```

7.2.2 Step 2: Start Local Container (on base station)

```
# macOS
cd docker/local
./start_mac.sh

# Linux
cd docker/local
./start_linux.sh
```

7.2.3 Step 3: Launch GUI

```
# Inside local container
source /opt/ros/humble/local_setup.bash
cd /app
python3 -m guis.gen_gui
```

7.3 Option B: Simulation Only

```
cd docker/local
./sim-launch.sh

# In another terminal (inside container)
ros2 launch my_robot_description display.launch.py use_sim_time:=true
```

7.4 Option C: Development Mode

```
# Start local container
cd docker/local
./start_linux.sh

# Inside container
source /opt/ros/humble/local_setup.bash
cd /app

# Run individual components
python3 ros_bridge/motor_bridge/launch_motor_bridge.py
python3 -m guis.arduino_gui
```

7.5 Quick Start Troubleshooting

If something isn't working, check these common issues first:

Table 7.1: Quick Start Troubleshooting Reference

Problem	Solution
Motors don't respond	**Re-plug Arduinos:** `docker-compose down` → unplug all → plug Sensor first → plug Motor second → `docker-compose up -d`
No sensor data (IMU/GPS)	Same as above — Sensor Arduino must be `/dev/ttyACM0` (plugged in first)
`/dev/ttyACM*` missing	Arduinos not plugged in, or USB cable issue. Try different USB port/cable
Container won't start	Check `docker ps -a` for error. Run `docker logs pitt_urc_jetson`
ROS topics not visible	Ensure `ROS_DOMAIN_ID=0` on both Jetson and base station
GUI won't display	Set `export DISPLAY=:0` or check X11 forwarding (XQuartz on macOS)

Important

90% of startup failures are caused by incorrect Arduino plug-in order. When in doubt:

1. Stop Docker: `sudo docker-compose down`
2. Unplug ALL Arduinos
3. Plug in Sensor Arduino (wait 2 sec)

4. Plug in Motor Arduino (wait 2 sec)
5. Verify: `ls /dev/ttyACM*` shows ACM0 and ACM1
6. Start Docker: `sudo docker-compose up -d`

For detailed troubleshooting, see Chapter 13.

Chapter 8

Detailed Startup Procedures

8.1 Jetson On-Board Startup

```
# 1. SSH into Jetson
ssh nvidia@<jetson-ip>

# 2. Navigate to project
cd /path/to/URC

# 3. Build and start container (first time)
cd docker/jetson
sudo docker build -t urc_jetson .
sudo docker-compose up -d

# 4. Verify container is running
sudo docker ps

# 5. Check supervised processes
sudo docker exec -it pitt_urc_jetson supervisorctl status

# Expected output:
# gps_bridge          RUNNING   pid 123, uptime 0:01:00
# motor_bridge        RUNNING   pid 124, uptime 0:01:00
# setup               RUNNING   pid 125, uptime 0:01:00
# ultrasonic_bridge   RUNNING   pid 126, uptime 0:01:00

# 6. View logs
sudo docker exec -it pitt_urc_jetson tail -f /app/motor_bridge.log
```

Listing 8.1: Complete Jetson Startup Procedure

8.2 Base Station Startup

8.2.1 macOS Procedure

```
# 1. Start XQuartz
open -a XQuartz

# 2. Enable network clients (if not done)
# XQuartz > Preferences > Security > Allow connections

# 3. Navigate to project
cd /path/to/URC/docker/local

```

```
# 4. Start container
./start_mac.sh

# 5. Source ROS 2
source /opt/ros/humble/local_setup.bash

# 6. Launch main GUI
cd /app
python3 -m guis.gen_gui
```

Listing 8.2: macOS Base Station Startup

8.2.2 Linux Procedure

```
# 1. Navigate to project
cd /path/to/URC/docker/local

# 2. Start container
./start_linux.sh

# 3. Source ROS 2
source /opt/ros/humble/local_setup.bash

# 4. Launch main GUI
cd /app
python3 -m guis.gen_gui
```

Listing 8.3: Linux Base Station Startup

8.3 Verification Steps

8.3.1 Check ROS 2 Connectivity

```
# List all active nodes
ros2 node list

# Expected:
# /motor_bridge
# /gps_bridge
# /imu_bridge
# /ultrasonic_bridge

# List all topics
ros2 topic list

# Expected:
# /cmd_vel
# /motor_control_input
# /drive_data
# /gps_data
# /imu_data
# /ultrasonic_data
# /camera/gray/image_raw
# /camera/depth/image_raw
```

Listing 8.4: ROS 2 Verification Commands

8.3.2 Test Motor Commands

Warning

Be careful when testing motor commands! Ensure the rover is in a safe position before sending movement commands.

```
# Send stop command
ros2 topic pub --once /motor_control_input std_msgs/String \
    "data: '0,0,0,0,0,0'"

# Send forward command (be careful!)
ros2 topic pub --once /motor_control_input std_msgs/String \
    "data: '100,0,0,0,0,0'"
```

Listing 8.5: Motor Command Testing

8.3.3 Monitor Sensor Data

```
# IMU data
ros2 topic echo /imu_data

# GPS data
ros2 topic echo /gps_data

# Ultrasonic data
ros2 topic echo /ultrasonic_data
```

Listing 8.6: Sensor Data Monitoring

Chapter 9

ROS 2 Architecture

9.1 Node Overview

Table 9.1: ROS 2 Nodes

Node	Package	Purpose
motor_bridge	ros_bridge	Motor command handling and feedback
gps_bridge	ros_bridge	GPS NMEA data publishing
imu_bridge	ros_bridge	IMU orientation data publishing
ultrasonic_bridge	ros_bridge	Distance measurement publishing
gray_and_depth_publisher	guis.camera	Camera feed publishing

9.2 Topic Architecture

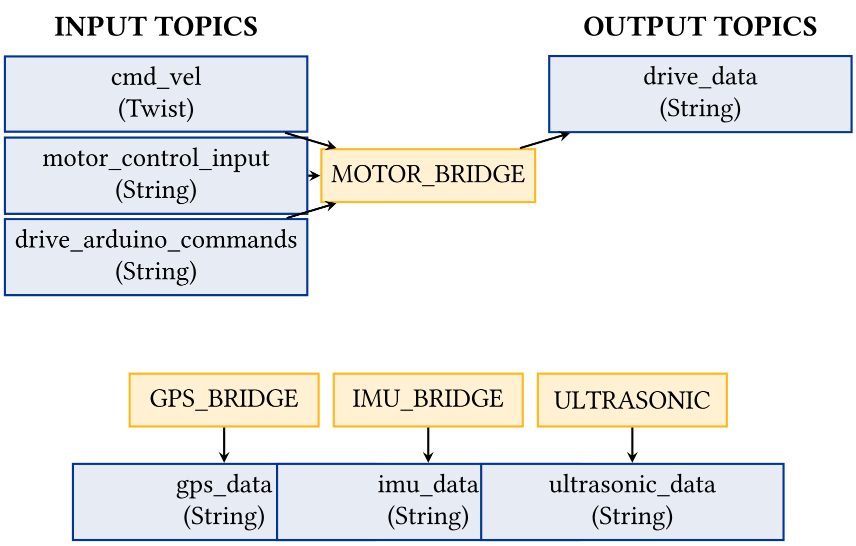

Figure 9.1: ROS 2 Topic Architecture

9.3 Message Formats

9.3.1 Motor Control Input (String)

```
Format: "linear_x,linear_y,linear_z,angular_x,angular_y,angular_z"
Example: "100,0,0,0,0,50"  # Forward with slight right turn
```

Listing 9.1: Motor Command Format

9.3.2 IMU Data (String)

```
Format: "X: <value>\tY: <value>\tZ: <value>"
Example: "X: 0.123\tY: 0.456\tZ: 0.789"
```

Listing 9.2: IMU Data Format

9.3.3 GPS Data (String)

```
NMEA sentences:
$GNGLL,4024.12345,N,07952.12345,W,123456.00,A,A*6B
$GAGSV,3,1,12,01,45,123,38,02,67,234,42,...
```

Listing 9.3: GPS NMEA Format

9.3.4 Ultrasonic Data (String)

```
Format: "distance1_cm, distance2_cm, distance3_cm, "
Example: "45, 120, 88, "
```

Listing 9.4: Ultrasonic Data Format

9.4 Bridge Class Hierarchy

```
ArduinoBridgeBase(Node)          # Base class (248 lines)
    |-- MotorBridge              # Motor control
    |-- GPSBridge                # GPS data
    |-- IMUBridge                # IMU data
    +-- UltrasonicBridge         # Ultrasonic data
```

Listing 9.5: Bridge Class Structure

Base Class Features:

- Serial port initialization (115200 baud)
- Generic ROS 2 topic subscription/publishing
- Timer-based Arduino reading (100ms default)
- Error handling and logging
- Automatic reconnection attempts

Chapter 10

Hardware Interfaces

10.1 Motor Control

10.1.1 Arduino Firmware

Location: `ros_bridge/motor_subscriber/motor_serial/motor_serial.ino`

```
// Simplified differential drive
left_speed = linear_x - angular_z;
right_speed = linear_x + angular_z;

// PWM mapping
pwm_left = map(left_speed, -100, 100, -255, 255);
pwm_right = map(right_speed, -100, 100, -255, 255);

// Direction Control
// HIGH = Forward
// LOW = Reverse
```

Listing 10.1: Motor Control Logic

10.2 IMU (BNO055)

Location: `ros_bridge/imu_bridge/imu_serial/imu_serial.ino`

```
Adafruit_BNO055 bno = Adafruit_BNO055(55);
bno.setExtCrystalUse(true);  // External crystal for accuracy

// Output Format
Serial.print("X: "); Serial.print(event.orientation.x);
Serial.print("\tY: "); Serial.print(event.orientation.y);
Serial.print("\tZ: "); Serial.println(event.orientation.z);
```

Listing 10.2: IMU Configuration

10.2.1 Data Processing

```
# IMUDataParser.py
import math

distance = math.sqrt(
    (x_cur - x_prev)**2 +
    (y_cur - y_prev)**2 +
    (z_cur - z_prev)**2
```

```
)
velocity = distance / time_delta
vertical_tilt = math.degrees(math.atan2(y_delta, z_delta))
horizontal_tilt = math.degrees(math.atan2(y_delta, x_delta))
```

Listing 10.3: IMU Data Parser

10.3 GPS

Location: ros_bridge/gps_bridge/gps/gps.ino

Table 10.1: NMEA Sentence Types

Type	Purpose
GNGLL	Geographic position (latitude/longitude)
GAGSV	GPS satellites in view + SNR
GBGSV	BeiDou satellites in view + SNR

10.4 Ultrasonic Sensors

Location: ros_bridge/ultrasonic_bridge/multiple_untrasonic_sensors/

```
duration = pulseIn(echoPin, HIGH);
distance_cm = (duration / 2) / 29.1;
```

Listing 10.4: Ultrasonic Distance Calculation

10.5 RealSense Cameras

Location: guis/camera/camera_cv_test.py

```python
import pyrealsense2 as rs

# Color stream
config.enable_stream(rs.stream.color, 640, 480, rs.format.bgr8, 30)

# Depth stream
config.enable_stream(rs.stream.depth, 640, 480, rs.format.z16, 30)
```

Listing 10.5: RealSense Configuration

Chapter 11

GUI System

11.1 Main Control GUI

File: `guis/gen_gui.py` (623 lines)

Table 11.1: Main GUI Components

Section	Function
Title Bar	Connection status (ONLINE/OFFLINE)
Navigation Tabs	Switch between competition modes
Camera Feeds	3 video displays (primary, secondary, auxiliary)
IMU Display	Speed, vertical tilt, horizontal tilt
System Controls	Toggle IMU, GPS, orientation
Emergency Stop	Red button for immediate halt

```
python3 -m guis.gen_gui
```

Listing 11.1: Launch Main GUI

11.2 Motor/Arm GUI Hotkeys

File: `guis/json_motorGUI.py` (292 lines)

Table 11.2: Motor Control Hotkeys

Key	Action
I	Forward
, (comma)	Backward
L	Turn Right
J	Turn Left
Q	Speed Up
Z	Slow Down
K	Stop

Table 11.3: Arm Control Hotkeys

Key	Action
0/9	Claw open/close
M/N	Base shift right/left
U/J	Bottom joint forward/backward
I/K	Middle joint forward/backward
O/L	Top joint forward/backward
Y/H	Wrist clockwise/counterclockwise
Escape	Emergency stop all

Warning

Keybinding Conflict Warning
The `json_motorGUI.py` Motor and Arm windows share overlapping key bindings that cause conflicts when both windows are open:

Key	Motor Action	Arm Action
I	Forward	Middle joint forward
J	Turn left	Bottom joint backward
K	Stop	Middle joint backward
L	Turn right	Top joint backward

Recommendation: Only open one control window at a time, or modify keybindings in the source code to eliminate conflicts.

11.3 Alternative GUI: HTTP-Based Control (json_motorGUI.py)

Location: `guis/json_motorGUI.py`

Important

Different Communication Architecture
Unlike the main GUIs that use ROS 2, the `json_motorGUI.py` uses HTTP POST requests to communicate with a local server. This requires a separate HTTP server running on `localhost:8000`.

Architecture:

1. GUI sends HTTP POST to `http://localhost:8000/`
2. JSON payload contains command code: `{"rover": 1}` or `{"arm": 0}`
3. HTTP server (not included) must translate commands to motor/arm control

Command Codes:

Table 11.4: Rover HTTP Command Codes

Code	Action	JSON Payload
1	Forward	`{"rover": 1}`
2	Reverse	`{"rover": 2}`
3	Turn Left	`{"rover": 3}`
4	Turn Right	`{"rover": 4}`
5	Speed Up (Boost)	`{"rover": 5}`
6	Slow Down	`{"rover": 6}`
7	Stop	`{"rover": 7}`

Table 11.5: Arm HTTP Command Codes

Code	Action	JSON Payload
0	Claw Open	`{"arm": 0}`
1	Claw Close	`{"arm": 1}`
2	Base Shift Right	`{"arm": 2}`
3	Base Shift Left	`{"arm": 3}`
4	Bottom Joint Forward	`{"arm": 4}`
5	Bottom Joint Reverse	`{"arm": 5}`
6	Middle Joint Forward	`{"arm": 6}`
7	Middle Joint Reverse	`{"arm": 7}`
8	Top Joint Forward	`{"arm": 8}`
9	Top Joint Reverse	`{"arm": 9}`
10	Wrist Clockwise	`{"arm": 10}`
11	Wrist Counterclockwise	`{"arm": 11}`
12	Stop All	`{"arm": 12}`

Warning

HTTP Server Requirement
The `json_motorGUI.py` will fail with connection errors if no HTTP server is running on port 8000. This GUI is designed for use with a custom HTTP-to-serial bridge that is separate from the ROS 2 infrastructure.

11.4 ROS 2 Motor Control GUI (arduino_gui.py)

Location: `guis/arduino_gui.py`

This GUI uses ROS 2 for motor control (unlike `json_motorGUI.py`).

Key Features:

- **ThrottleControl Widget** — Custom QFrame with slider (0-100%), reset/full buttons
- **Physics Simulation** — Acceleration/deceleration at 20 FPS (50ms timer)
- **WASD Controls** — Hold SHIFT + direction keys to apply throttle
- **ROS 2 Integration** — Publishes to `motor_control_input` topic via MotorPublisher

Keybindings (arduino_gui.py):

Key	Action
W	Forward (with SHIFT held)
S	Backward (with SHIFT held)
A	Turn Left (with SHIFT held)
D	Turn Right (with SHIFT held)
SHIFT	Apply throttle (hold to accelerate)
SPACE	Emergency brake (immediate stop)

Physics Parameters:

```
self.acceleration_rate = 0.1      # Rate toward target speed
self.deceleration_rate = 0.08     # Rate when releasing throttle
self.angular_deceleration_rate = 0.1
# Angular velocity decay multiplier: 0.7 (applied per frame)
# Physics update rate: 50ms (20 FPS)
```

Listing 11.2: Physics Constants in arduino_gui.py

11.5 Publisher Classes

Location: `guis/publishers/publisher.py`

```
class MotorPublisher(GenericPublisher):
    def __init__(self):
        super().__init__('motor_control_input', String)

    def publish_motor_command(self, motor_values):
        # motor_values: list of 6 floats
        msg = String()
        msg.data = ','.join(map(str, motor_values))
        self.publisher.publish(msg)

    def stop_all_motors(self):
        self.publish_motor_command([0, 0, 0, 0, 0, 0])
```

Listing 11.3: MotorPublisher Class

```
class TwistPublisher(GenericPublisher):
    def __init__(self):
        super().__init__('cmd_vel', Twist)

    def move_forward(self, speed):
        msg = Twist()
        msg.linear.x = float(speed)
        self.publisher.publish(msg)

    def turn_left(self, angular_speed):
        msg = Twist()
        msg.angular.z = float(angular_speed)
        self.publisher.publish(msg)
```

Listing 11.4: TwistPublisher Class

11.6 Event Handling Architecture

The GUI system uses PyQt5's signal/slot mechanism to safely communicate between ROS 2 threads and the GUI thread.

11.6.1 Signal/Slot Pattern

ROS 2 callbacks execute in separate threads, but Qt GUI updates must happen in the main thread. Signals bridge this gap:

```python
# In IMUSubscriber (guis/subscribers/subscriber.py)
class IMUSubscriber(GenericSubscriber, QObject):
    # Define signal with parameter types
    imu_data_updated = pyqtSignal(float, float, float)

    def default_callback(self, msg):
        # Parse data in ROS thread
        parsed_data = self.parser.parse_imu_data(msg.data)
        self.velocity, self.vertical_tilt, self.horizontal_tilt = parsed_data[1:4]

        # Emit signal (thread-safe crossing to GUI thread)
        self.imu_data_updated.emit(
            self.velocity,
            self.vertical_tilt,
            self.horizontal_tilt
        )
```

Listing 11.5: Signal Definition and Emission

```python
# In MainWindow (guis/gen_gui.py)
def setup_subscribers(self):
    self.imu_subscriber = IMUSubscriber()

    # Connect signal to slot (GUI update method)
    self.imu_subscriber.imu_data_updated.connect(self.update_imu_display)

def update_imu_display(self, velocity, vert_tilt, horiz_tilt):
    # This runs in GUI thread - safe to update widgets
    self.speed_label.setText(f"{velocity:.2f} m/s")
    self.vert_tilt_label.setText(f"{vert_tilt:.1f} deg")
    self.horiz_tilt_label.setText(f"{horiz_tilt:.1f} deg")
```

Listing 11.6: Signal Connection in GUI

11.6.2 Available Signals

Table 11.6: PyQt5 Signals in the System

Class	Signal	Parameters
IMUSubscriber	imu_data_updated	(float, float, float) — velocity, vert_tilt, horiz_tilt
MotorPublisher	message_published	(str) — the published command string
TwistPublisher	twist_published	(float × 6) — linear xyz, angular xyz

11.6.3 Keyboard Event Handling

The `json_motorGUI.py` implements keyboard control using Qt's `keyPressEvent`:

```python
# Command codes (from json_motorGUI.py lines 9-30)
FORWARD = 1
REVERSE = 2
```

```python
LEFT = 3
RIGHT = 4
SPEED_UP = 5
SLOW_DOWN = 6
STOP = 7

class MotorWindow(QWidget):
    def keyPressEvent(self, event):
        key = event.key()

        if key == Qt.Key_I:
            self.send_command(FORWARD)
        elif key == Qt.Key_Comma:
            self.send_command(REVERSE)
        elif key == Qt.Key_J:
            self.send_command(LEFT)
        elif key == Qt.Key_L:
            self.send_command(RIGHT)
        elif key == Qt.Key_K:
            self.send_command(STOP)
        elif key == Qt.Key_Escape:
            self.emergency_stop()
```

Listing 11.7: Keyboard Control Pattern

11.7 Extending the GUI System

11.7.1 Adding a New Sensor Display

To add a new sensor type to the GUI:

1. **Create Data Parser** (if needed) in `guis/subscribers/`
2. **Create Subscriber Class** inheriting from `GenericSubscriber`
3. **Define PyQt Signal** with appropriate parameter types
4. **Connect Signal to Slot** in your GUI class
5. **Create UI Widgets** to display the data

```python
# 1. Create TempSubscriber in guis/subscribers/subscriber.py
class TempSubscriber(GenericSubscriber, QObject):
    temp_updated = pyqtSignal(float)  # temperature in Celsius

    def __init__(self):
        super().__init__('temp_data', String, 'temp_subscriber')
        QObject.__init__(self)

    def default_callback(self, msg):
        temp = float(msg.data)
        self.temp_updated.emit(temp)

# 2. In your GUI class
self.temp_subscriber = TempSubscriber()
self.temp_subscriber.temp_updated.connect(self.update_temp_display)

def update_temp_display(self, temp):
    self.temp_label.setText(f"Temp: {temp:.1f} C")
```

Listing 11.8: Example: Adding Temperature Sensor Display

11.7.2 Adding a New Control Button

To add a new control that sends commands to the rover:

```python
# In your GUI setup
self.custom_button = QPushButton("Custom Action")
self.custom_button.clicked.connect(self.on_custom_click)

# Create publisher instance
self.motor_pub = MotorPublisher()

def on_custom_click(self):
    # Send custom motor command
    self.motor_pub.publish_motor_command([50, 0, 0, 0, 0, 25])
```

Listing 11.9: Adding Custom Control Button

Chapter 12

Development Workflow

12.1 Git Workflow

12.1.1 Branch Strategy

- `main` — Protected, requires PR review
- `feature/*` — New features
- `bugfix/*` — Bug fixes
- `hotfix/*` — Urgent production fixes

12.1.2 Standard Workflow

```
# 1. Fetch latest
git fetch origin

# 2. Create feature branch
git checkout -b feature/my-feature main

# 3. Make changes
# ... edit files ...

# 4. Stage and commit
git add .
git commit -m "feat: add my feature description"

# 5. Push to remote
git push origin feature/my-feature

# 6. Create Pull Request on GitHub
```

Listing 12.1: Git Development Workflow

12.1.3 Commit Message Format

```
type: description

Types:
- feat: New feature
- fix: Bug fix
- docs: Documentation
- refactor: Code refactoring
```

```
- test: Testing
- chore: Maintenance
```

Listing 12.2: Commit Message Types

12.2 Adding a New Sensor Bridge

12.2.1 Step 1: Create Arduino Firmware

```
// ros_bridge/new_sensor_bridge/new_sensor_serial/new_sensor_serial.ino
void setup() {
    Serial.begin(115200);
}

void loop() {
    // Read sensor
    float value = readSensor();

    // Send data
    Serial.println(value);
    delay(100);
}
```

Listing 12.3: New Sensor Arduino Code

12.2.2 Step 2: Create Python Bridge

```
# ros_bridge/new_sensor_bridge/new_sensor_bridge.py
from ros_bridge.arduino_bridge_base.arduino_bridge_base import ArduinoBridgeBase
from std_msgs.msg import String

class NewSensorBridge(ArduinoBridgeBase):
    def __init__(self):
        super().__init__(
            node_name='new_sensor_bridge',
            topic_name='new_sensor_data',
            msg_type=String,
            serial_port='/dev/ttyACM0',
            baud_rate=115200
        )
```

Listing 12.4: New Sensor Python Bridge

12.2.3 Step 3: Add to Supervisor

```
# docker/jetson/supervisord.conf
[program:new_sensor_bridge]
command=python3 /app/ros_bridge/new_sensor_bridge/new_sensor_bridge.py
stdout_logfile=/app/new_sensor_bridge.log
stderr_logfile=/app/new_sensor_bridge_err.log
autorestart=true
```

Listing 12.5: Supervisor Configuration Entry

Chapter 13

Troubleshooting

Important

#1 Cause of System Failures: Arduino Plug-In Order
Before troubleshooting anything else, verify:

1. Did you **re-plug Arduinos before starting Docker**?
2. Was **Sensor Arduino plugged in FIRST** (becomes `/dev/ttyACM0`)?
3. Was **Motor Arduino plugged in SECOND** (becomes `/dev/ttyACM1`)?

Docker captures port mappings at startup. If you started the container with Arduinos in the wrong order (or unplugged), the bridges will fail silently or connect to the wrong device.
Quick Fix: `docker-compose down` → unplug all → plug Sensor → plug Motor → `docker-compose up -d`
See Section 13.2.3 for the complete USB Port Enumeration Order guide.

13.1 Common Issues

Issue	Cause	Solution
Serial port not found	Device not connected or wrong plug-in order	Check USB connections, verify `/dev/ttyACM*` exists. **Ensure correct plug-in order** (see Section 13.2.3)
Motors not responding	Wrong port, baud rate, or plug-in order	Motor Arduino must be `/dev/ttyACM1` (plugged in **second**). See Section 13.2.3 for required order
Sensor data missing	Wrong plug-in order	Sensor Arduino must be `/dev/ttyACM0` (plugged in **first**). See Section 13.2.3
ROS topics not visible	Domain ID mismatch	Ensure `ROS_DOMAIN_ID=0` on both systems

Issue	Cause	Solution
GUI won't display	X11 forwarding issue	Check `DISPLAY` variable, restart XQuartz
Camera black screen	USB bandwidth	Use USB 3.0 ports, reduce resolution
High latency	Network congestion	Check WiFi signal, reduce message frequency
IMU returns -0.1 values	Parse error in IMUDataParser	Check serial format matches "X: val\tY: val\tZ: val", verify BNO055 calibration
GPS shows "NO CONNECTION!!!"	Empty NMEA sentence	Check GPS serial connection, verify module has satellite fix
Motor command ignored	Invalid 6-value format	Verify comma-separated format with no spaces, check for exactly 6 values
Ultrasonic shows 0.0	Sensor timeout or wiring	Check trigger/echo pin wiring, verify object is within sensor range (2-400cm)
Qt platform plugin error	Missing X11 libraries	Install `libxcb-xinerama0`, restart container
Camera black after restart	RealSense USB reset needed	Unplug/replug USB cable, run `rs-enumerate-devices`
ROS nodes visible but no data	QoS mismatch	Ensure `queue_size=10` on both publisher and subscriber
Container exits immediately	Supervisor config error	Check `/app/*.log` files inside container for startup errors
PyQt5 signal not received	Thread safety violation	Use `pyqtSignal` for cross-thread communication, never update GUI directly from ROS callback
Serial permission denied	User not in dialout group	Run `sudo usermod -a -G dialout $USER`, then logout/login

Table 13.1: Common Issues and Solutions

13.2 Docker Container Restart Guide

When serial ports change (e.g., Arduino reconnected to different USB port), you must restart the Docker container to pick up the new device mapping.

Important

Key Concept: Docker Captures Ports at Startup
The Docker container maps `/dev/ttyACM*` devices **only when it starts**. If you:

- Unplug/replug an Arduino while the container is running → container won't see the change

- Start the container with Arduinos unplugged → container has no serial access
- Start the container with wrong plug-in order → ports are swapped, motors/sensors fail

Solution: Always re-plug Arduinos in correct order **before** every `docker-compose up`.

13.2.1 When to Restart

- Arduino disconnected and reconnected (port may change from `/dev/ttyACM0` to `/dev/ttyACM1`)
- New USB device added that wasn't present at container start
- Serial permission changes made to host system
- After modifying `docker-compose.yml` device mappings
- **Every time you want fresh, correct serial port assignments**

13.2.2 Complete Restart Procedure

```
# 1. Stop the running container
cd docker/jetson  # or docker/local
sudo docker-compose down

# 2. RE-PLUG ARDUINOS IN CORRECT ORDER (CRITICAL!)
#    a. Unplug ALL Arduinos from Jetson
#    b. Wait 2 seconds
#    c. Plug in SENSOR Arduino FIRST -> becomes /dev/ttyACM0
#    d. Wait 2 seconds
#    e. Plug in MOTOR Arduino SECOND -> becomes /dev/ttyACM1

# 3. Verify correct port assignments
ls -la /dev/ttyACM*
# MUST show: ttyACM0 (Sensor) and ttyACM1 (Motor)

# 4. (Optional) Update docker-compose.yml if needed
# Edit the devices section if needed:
#   devices:
#     - /dev/ttyACM0:/dev/ttyACM0
#     - /dev/ttyACM1:/dev/ttyACM1

# 5. Start the container (AFTER re-plugging Arduinos!)
sudo docker-compose up -d

# 6. Verify container is running and processes started
sudo docker ps
sudo docker exec -it pitt_urc_jetson supervisorctl status

# 7. Check bridge logs for serial connection success
sudo docker exec -it pitt_urc_jetson tail -f /app/motor_bridge.log
# Look for: "Serial connection established on /dev/ttyACM1"
```

Listing 13.1: Docker Container Restart for Port Changes

Warning

Port Hardcoding Warning
The `MotorBridge` class has the serial port hardcoded to `/dev/ttyACM1`. If your Arduino appears on a different port, you have two options:

1. **Create symlink on host** (recommended):

```
sudo ln -sf /dev/ttyACM0 /dev/ttyACM1

```

2. **Modify source code**: Edit `ros_bridge/arduino_bridge_base/arduino_bridge_base.py` line 98:

```
serial_port="/dev/ttyACM0",  # Change to your port

```

13.2.3 USB Port Enumeration Order

Important

Critical: Arduino Plug-In Order Matters
The Jetson (and Linux in general) assigns serial port numbers based on the **order devices are plugged in**, not by physical USB port location. The first USB serial device plugged in becomes `/dev/ttyACM0`, the second becomes `/dev/ttyACM1`, and so on.

How Port Assignment Works:

1. When the Jetson boots with no USB devices connected, no `/dev/ttyACM*` ports exist
2. The **first Arduino plugged in** is assigned `/dev/ttyACM0`
3. The **second Arduino plugged in** is assigned `/dev/ttyACM1`
4. If a device is unplugged and replugged, it may get a different port number

Required Plug-In Order for URC System:
Based on the hardcoded port assignments in `arduino_bridge_base.py`:

Order	Arduino	Expected Port	Bridge Class
1st (plug in first)	Sensor Arduino (IMU/GPS/Ultrasonic)	`/dev/ttyACM0`	IMUBridge, GPSBridge, UltrasonicB
2nd (plug in second)	Motor Arduino	`/dev/ttyACM1`	MotorBridge

```
# MotorBridge - HARDCODED to ACM1 (line 98)
class MotorBridge(ArduinoBridgeBase):
    def __init__(self):
        super().__init__(
            serial_port="/dev/ttyACM1",  # Must be second device
            ...
        )

# Sensor bridges default to ACM0
class UltrasonicBridge(ArduinoBridgeBase):
    def __init__(self, serial_port="/dev/ttyACM0", ...):  # First device
```

```python

class GPSBridge(ArduinoBridgeBase):
    def __init__(self, serial_port="/dev/ttyACM0", ...):  # First device

class IMUBridge(ArduinoBridgeBase):
    def __init__(self):
        possible_ports = ["/dev/ttyACM0", ...]  # Tries ACM0 first
```

Listing 13.2: Port Assignments in arduino_bridge_base.py

Startup Procedure:

1. Power on Jetson with **no Arduinos connected**
2. Wait for Jetson to fully boot (login screen or SSH available)
3. Plug in **Sensor Arduino first** (IMU/GPS/Ultrasonic) → becomes `/dev/ttyACM0`
4. Plug in **Motor Arduino second** → becomes `/dev/ttyACM1`
5. Verify ports with: `ls -la /dev/ttyACM*`
6. Start Docker container: `sudo docker-compose up -d`

Warning

Common Mistake: Wrong Plug-In Order
If you plug in the Motor Arduino first, it gets `/dev/ttyACM0` but the MotorBridge expects `/dev/ttyACM1`. Symptoms:

- Motor commands have no effect
- Bridge logs show "Serial connection established" but motors don't respond
- IMU/GPS bridge fails to connect (expected port is taken by wrong Arduino)

Fix: Unplug both Arduinos, wait 2 seconds, then plug them in the correct order.

Verifying Correct Port Assignment:

```bash
# List connected serial devices
ls -la /dev/ttyACM*

# Expected output when correctly plugged in:
# /dev/ttyACM0 -> Sensor Arduino (first plugged in)
# /dev/ttyACM1 -> Motor Arduino (second plugged in)

# Check which Arduino is on which port by monitoring serial output:
# Terminal 1 - Monitor ACM0 (should show IMU/GPS data)
screen /dev/ttyACM0 115200

# Terminal 2 - Monitor ACM1 (should respond to motor commands)
screen /dev/ttyACM1 115200

# Exit screen: Ctrl+A, then K, then Y
```

Listing 13.3: Verify Arduino Port Assignment

Tip

Pro Tip: udev Rules for Persistent Naming
For production deployments, create udev rules to assign consistent device names based on hardware serial numbers instead of plug-in order:

```
# Find Arduino serial number
udevadm info -a -n /dev/ttyACM0 | grep serial

# Create udev rule (as root)
echo 'SUBSYSTEM=="tty", ATTRS{serial}=="YOUR_SERIAL", SYMLINK+="arduino_motor"' \
    >> /etc/udev/rules.d/99-arduino.rules

# Reload rules
sudo udevadm control --reload-rules
```

Then update bridge code to use `/dev/arduino_motor` instead of `/dev/ttyACM1`.

13.2.4 Quick Restart Commands

Warning

Remember: Before ANY `docker-compose up` command, re-plug Arduinos in correct order!

```
# === BEFORE ANY RESTART: Re-plug Arduinos ===
# 1. Unplug all Arduinos
# 2. Plug Sensor Arduino (ACM0)
# 3. Plug Motor Arduino (ACM1)
# 4. Verify: ls /dev/ttyACM*

# Full restart (recommended for port changes)
sudo docker-compose down && sudo docker-compose up -d

# Restart single bridge process (port must be same - NO re-plug needed)
sudo docker exec -it pitt_urc_jetson supervisorctl restart motor_bridge

# Rebuild and restart (after code changes)
sudo docker-compose down
# Re-plug Arduinos here!
sudo docker build -t urc_jetson .
sudo docker-compose up -d

# Force recreate container (clears all state)
# Re-plug Arduinos first!
sudo docker-compose up -d --force-recreate
```

Listing 13.4: Quick Restart Reference

13.3 Complete Cold-Start Procedure

This section provides a comprehensive, step-by-step guide for starting the entire URC system from a powered-off state.

Important

Critical: Follow This Order Exactly

The USB port enumeration is determined by plug-in order. Deviating from this procedure will result in incorrect port assignments and system failures.

13.3.1 Phase 1: Hardware Power-Up

1. **Ensure all Arduinos are UNPLUGGED** from the Jetson
2. Power on the Jetson Nano
3. Wait for complete boot (login prompt or SSH available) — approximately 60-90 seconds
4. Power on the base station computer

13.3.2 Phase 2: Arduino Connection (Critical Order)

1. **Plug in Sensor Arduino FIRST**
   - This Arduino runs IMU, GPS, and ultrasonic firmware
   - It will be assigned `/dev/ttyACM0`
   - Wait 2 seconds for device enumeration
2. **Plug in Motor Arduino SECOND**
   - This Arduino runs motor control firmware
   - It will be assigned `/dev/ttyACM1`
   - Wait 2 seconds for device enumeration
3. **Verify port assignments on Jetson:**

```
ls -la /dev/ttyACM*
# Expected output:
# /dev/ttyACM0 -> Sensor Arduino
# /dev/ttyACM1 -> Motor Arduino

```

13.3.3 Phase 3: Start Jetson Container

```
# SSH into Jetson (or use direct terminal)
ssh user@jetson-ip

# Navigate to docker directory
cd ~/URC/docker/jetson

# Start container in detached mode
sudo docker-compose up -d

# Verify container is running
sudo docker ps
# Should show: pitt_urc_jetson

# Check Supervisor processes started
sudo docker exec -it pitt_urc_jetson supervisorctl status
# Expected: motor_bridge RUNNING, imu_bridge RUNNING, etc.

# Monitor bridge logs for serial connection success
sudo docker exec -it pitt_urc_jetson tail -f /app/motor_bridge.log
# Look for: "Serial connection established on /dev/ttyACM1"
```

Listing 13.5: Jetson Container Startup

13.3.4 Phase 4: Start Base Station Container

```
# On base station computer
cd ~/URC/docker/local

# Start container
sudo docker-compose up -d

# Verify container
sudo docker ps
# Should show: pitt_urc_local
```

Listing 13.6: Base Station Container Startup

13.3.5 Phase 5: Verify ROS 2 Communication

```
# On base station, enter container
sudo docker exec -it pitt_urc_local bash

# Source ROS 2
source /opt/ros/humble/local_setup.bash

# Check for topics from Jetson
ros2 topic list
# Expected: /imu_data, /gps_data, /motor_control_input, /cmd_vel, etc.

# Test IMU data reception
ros2 topic echo /imu_data
# Should show: data: "X: <val>\tY: <val>\tZ: <val>"

# Test motor command transmission
ros2 topic pub --once /motor_control_input std_msgs/String "data: '0,0,0,0,0,0'"
# Motors should remain stopped (all zeros)
```

Listing 13.7: ROS 2 Verification

13.3.6 Phase 6: Launch GUI

```
# Inside base station container
cd /app/guis

# Launch main GUI
python3 gen_gui.py

# OR launch motor control GUI (ROS 2 based)
python3 arduino_gui.py

# OR launch HTTP-based GUI (requires HTTP server)
# python3 json_motorGUI.py
```

Listing 13.8: GUI Launch

13.3.7 Verification Checklist

Table 13.2: Cold-Start Verification Checklist

Component	Status	Verification Command
Jetson booted	□	SSH connection successful
Sensor Arduino (ACM0)	□	`ls /dev/ttyACM0` exists
Motor Arduino (ACM1)	□	`ls /dev/ttyACM1` exists
Jetson container	□	`docker ps` shows running
Bridges running	□	`supervisorctl status` all RUNNING
Serial connected	□	Bridge logs show "established"
Base container	□	`docker ps` shows running
ROS topics visible	□	`ros2 topic list` shows topics
IMU data flowing	□	`ros2 topic echo /imu_data` shows data
GUI launches	□	Window appears, no errors

Warning

Troubleshooting Cold-Start Failures

- **No /dev/ttyACM* devices:** Unplug both Arduinos, wait 5 seconds, replug in correct order
- **Bridge shows "Serial connection failed":** Wrong plug-in order. Restart from Phase 2
- **ROS topics not visible:** Check `ROS_DOMAIN_ID=0` on both systems
- **GUI crashes on launch:** Verify X11 forwarding (`echo $DISPLAY` should be set)
- **Motors don't respond:** Verify Motor Arduino is on ACM1 (plug-in order issue)

13.4 Serial Port Debugging

```
# List all serial ports
ls -la /dev/ttyACM* /dev/ttyUSB*

# Check port permissions
sudo chmod 666 /dev/ttyACM0

# Test serial communication
screen /dev/ttyACM0 115200

# Kill screen session: Ctrl+A, then K
```

Listing 13.9: Serial Port Debugging Commands

13.5 ROS 2 Debugging

```
# Check ROS environment
echo $ROS_DOMAIN_ID
```

```bash
echo $ROS_LOCALHOST_ONLY

# Source ROS 2
source /opt/ros/humble/local_setup.bash

# Verify discovery
ros2 daemon status
ros2 daemon start

# Check multicast
ros2 multicast receive
```

Listing 13.10: ROS 2 Debugging Commands

13.6 Docker Debugging

```bash
# Check container status
docker ps -a

# View container logs
docker logs pitt_urc_jetson

# Enter running container
docker exec -it pitt_urc_jetson bash

# Check supervisor status (inside Jetson container)
supervisorctl status
supervisorctl restart motor_bridge
```

Listing 13.11: Docker Debugging Commands

13.7 Log File Locations

Table 13.3: Log File Locations

Log	Location	Content
Motor Bridge	/app/motor_bridge.log	Motor commands, errors
GPS Bridge	/app/gps_bridge.log	NMEA sentences
Ultrasonic	/app/ultrasonic_bridge.log	Distance readings
Setup	/app/setup.log	Camera initialization

Chapter 14

API Reference

14.1 ArduinoBridgeBase

```python
class ArduinoBridgeBase(Node):
    """Base class for all Arduino serial bridges."""

    def __init__(self, node_name, topic_name, msg_type,
                 serial_port='/dev/ttyACM0', baud_rate=115200):
        """
        Initialize the bridge.

        Args:
            node_name: ROS 2 node name
            topic_name: Topic to publish/subscribe
            msg_type: ROS 2 message type (e.g., String)
            serial_port: Serial port path
            baud_rate: Baud rate (default: 115200)
        """

    def read_from_arduino(self):
        """Read data from Arduino and publish to ROS topic."""

    def write_to_arduino(self, data):
        """Write data to Arduino serial port."""
```

Listing 14.1: ArduinoBridgeBase API

14.2 MotorPublisher

```python
class MotorPublisher(GenericPublisher):
    """Publisher for motor control commands."""

    def publish_motor_command(self, motor_values: List[float]):
        """
        Publish motor command.

        Args:
            motor_values: List of 6 float values
                          [linear_x, linear_y, linear_z,
                           angular_x, angular_y, angular_z]
        """

    def set_motor_value(self, index: int, value: float):
        """Set a specific motor value (0-5)."""
```

```python

    def set_all_motors(self, value: float):
        """Set all motors to the same value."""

    def stop_all_motors(self):
        """Emergency stop - set all motors to 0."""
```

Listing 14.2: MotorPublisher API

14.3 TwistPublisher

```python
class TwistPublisher(GenericPublisher):
    """Publisher for geometry_msgs/Twist messages."""

    def publish_twist(self, linear_x=0, linear_y=0, linear_z=0,
                      angular_x=0, angular_y=0, angular_z=0):
        """Publish a full Twist message."""

    def move_forward(self, speed: float):
        """Move forward at specified speed."""

    def move_backward(self, speed: float):
        """Move backward at specified speed."""

    def turn_left(self, angular_speed: float):
        """Turn left at specified angular speed."""

    def turn_right(self, angular_speed: float):
        """Turn right at specified angular speed."""

    def stop(self):
        """Stop all motion."""
```

Listing 14.3: TwistPublisher API

14.4 GenericPublisher

Location: `guis/publishers/publisher.py`

Base class for all ROS 2 publishers in the GUI system.

```python
class GenericPublisher(Node):
    """
    Base class for ROS 2 publishers.

    Extends rclpy.node.Node to provide a simple interface
    for publishing messages to ROS 2 topics.
    """

    def __init__(self, topic_name: str, msg_type,
                 node_name: str = 'generic_publisher',
                 queue_size: int = 10):
        """
        Initialize the publisher.

        Args:
            topic_name: Name of the ROS 2 topic to publish to
            msg_type: Message type (e.g., String, Twist)
            node_name: Unique name for this ROS 2 node
            queue_size: Publisher queue depth (default: 10)
```

```
        """

    def publish_message(self, message):
        """
        Publish a message to the topic.

        Args:
            message: Message instance of the correct type
        """
```

Listing 14.4: GenericPublisher API

14.5 GenericSubscriber

Location: `guis/subscribers/subscriber.py`

Base class for all ROS 2 subscribers in the GUI system.

```
class GenericSubscriber(Node):
    """
    Base class for ROS 2 subscribers.

    Provides subscription to ROS 2 topics with customizable
    callback functions for message processing.
    """

    def __init__(self, topic_name: str, msg_type,
                 node_name: str = 'generic_subscriber',
                 callback=None):
        """
        Initialize the subscriber.

        Args:
            topic_name: Name of the ROS 2 topic to subscribe to
            msg_type: Expected message type
            node_name: Unique name for this ROS 2 node
            callback: Custom callback function (uses default if None)
        """

    def default_callback(self, msg):
        """
        Default message handler - logs received messages.

        Override in subclasses for custom processing.

        Args:
            msg: The received ROS 2 message
        """
```

Listing 14.5: GenericSubscriber API

14.6 IMUSubscriber

Location: `guis/subscribers/subscriber.py`

Specialized subscriber for IMU data with PyQt5 signal integration.

```
class IMUSubscriber(GenericSubscriber, QObject):
    """
    Subscriber for IMU data with Qt signal support.

    Parses incoming IMU messages and emits signals for
```

```
    thread-safe GUI updates.
    """

    # PyQt5 Signal: emitted when new IMU data is parsed
    imu_data_updated = pyqtSignal(float, float, float)
    # Parameters: velocity, vertical_tilt, horizontal_tilt

    def __init__(self, topic_name: str = 'imu_data',
                 msg_type=String,
                 node_name: str = 'imu_subscriber'):
        """Initialize with default IMU topic."""

    @property
    def imu_distance(self) -> float:
        """Cumulative distance traveled (meters)."""

    @property
    def imu_velocity(self) -> float:
        """Current velocity (meters/second)."""

    @property
    def imu_vertical_tilt_angle(self) -> float:
        """Vertical tilt angle (degrees)."""

    @property
    def imu_horizontal_tilt_angle(self) -> float:
        """Horizontal tilt angle (degrees)."""

    # Error Behavior:
    # On parse failure, all properties return -0.1
```

Listing 14.6: IMUSubscriber API

14.7 GPSSubscriber

Location: `guis/subscribers/subscriber.py`

Specialized subscriber for GPS NMEA data.

```
class GPSSubscriber(GenericSubscriber):
    """
    Subscriber for GPS NMEA sentence data.

    Parses GAGSV/GBGSV (satellite) and GNGLL (position)
    sentences using GPSDataParser.
    """

    def __init__(self, topic_name: str = 'gps_data',
                 msg_type=String,
                 node_name: str = 'gps_subscriber'):
        """Initialize with default GPS topic."""

    def default_callback(self, msg):
        """
        Parse GPS message and log results.

        Logs: SNR values, average SNR, coordinates
        Returns "NO CONNECTION!!!" on empty input
        """
```

Listing 14.7: GPSSubscriber API

14.8 IMUDataParser

Location: `guis/subscribers/IMUDataParser.py`

Stateful parser that computes derived values from IMU orientation data.

```python
class IMUDataParser:
    """
    Parses IMU orientation strings and computes motion metrics.

    Maintains state between readings to calculate distance
    and velocity. Uses Euclidean distance in 3D space.
    """

    def __init__(self):
        """Initialize with no previous data."""

    def parse_imu_data(self, imu_message: str) -> tuple:
        """
        Parse IMU message and compute derived values.

        Args:
            imu_message: String in format "X: <f>\tY: <f>\tZ: <f>"

        Returns:
            tuple: (distance, velocity, vert_tilt, horiz_tilt)
                   Returns (0.0, 0.0, 0.0, 0.0) on first call
        """

    @staticmethod
    def parse_coordinates(imu_coordinates_data: str) -> list[float]:
        """
        Extract [X, Y, Z] floats from IMU string.

        Args:
            imu_coordinates_data: "X: val\tY: val\tZ: val"

        Returns:
            list: [x, y, z] as floats
        """

    @staticmethod
    def calculate_distance(past: list, current: list) -> float:
        """
        Compute Euclidean distance between 3D points.

        Args:
            past: [x, y, z] previous coordinates
            current: [x, y, z] current coordinates

        Returns:
            float: Distance in coordinate units
        """
```

Listing 14.8: IMUDataParser API

14.9 GPSDataParser

Location: `guis/subscribers/GPSDataParser.py`

Parser for NMEA GPS sentences (GAGSV, GBGSV, GNGLL).

```python
class GPSDataParser:
```

```
    """
    Parses NMEA sentences for GPS position and signal quality.

    Supports:
    - GAGSV/GBGSV: Satellite SNR (signal-to-noise ratio)
    - GNGLL: Geographic position (lat/lon)
    """

    def parse_gagsv(self, nmea_sentence: str) -> list[int] | str | None:
        """
        Extract SNR values from satellite sentence.

        Args:
            nmea_sentence: NMEA GAGSV or GBGSV sentence

        Returns:
            list[int]: SNR values for visible satellites
            str: "NO CONNECTION!!!" if empty input
            None: If not a valid GAGSV/GBGSV sentence
        """

    def compute_average_snr(self, snr_values: list[int]) -> float:
        """
        Calculate average signal-to-noise ratio.

        Args:
            snr_values: List of SNR integers

        Returns:
            float: Average SNR, or 0 if empty list
        """

    def parse_gngll(self, nmea_sentence: str) -> str | None:
        """
        Extract coordinates from position sentence.

        Args:
            nmea_sentence: NMEA GNGLL sentence

        Returns:
            str: "Latitude: <lat> <dir> Longitude: <lon> <dir>"
            str: "NO CONNECTION!!!" if empty input
            None: If not a valid GNGLL sentence
        """

    def parse_message(self, nmea_sentence: str) -> tuple:
        """
        Parse any supported NMEA sentence.

        Args:
            nmea_sentence: Any NMEA sentence string

        Returns:
            tuple: (snr_values, average_snr, coordinates)
                   - snr_values: list, "NO CONNECTION!!!", or None
                   - average_snr: float (0 if no SNR data)
                   - coordinates: str or None
        """
```

Listing 14.9: GPSDataParser API

14.10 ImageSubscriber

Location: `guis/subscribers/RGBsubcriber.py`

Subscriber for RealSense camera images (grayscale and depth).

```python
class ImageSubscriber(Node):
    """
    Subscribes to RealSense camera image topics.

    Handles both grayscale and depth images using CvBridge
    for ROS-to-OpenCV conversion.

    Topics:
        - camera/gray/image_raw: Grayscale images (mono8)
        - camera/depth/image_raw: Depth images (passthrough)
    """

    def __init__(self):
        """
        Initialize subscriptions for gray and depth images.

        Creates CvBridge for image format conversion.
        """

    def gray_callback(self, msg):
        """
        Process incoming grayscale images.

        Args:
            msg (sensor_msgs/Image): ROS image message

        Side Effects:
            - Updates self.gray_image (numpy.ndarray)
            - Displays image via cv2.imshow() if enabled

        Encoding: mono8 (8-bit grayscale)
        """

    def depth_callback(self, msg):
        """
        Process incoming depth images.

        Args:
            msg (sensor_msgs/Image): ROS depth image message

        Encoding: passthrough (native depth format)

        Note: Display visualization commented out in production.
              Uncomment cv2.convertScaleAbs() lines for
              colormap visualization.
        """

# Utility function
def array_to_image(array, filename) -> PILImage:
    """
    Convert numpy array to PIL Image and save to file.

    Args:
        array: 2D numpy array (clipped to 0-255)
        filename: Output file path

    Returns:
        PIL.Image: Grayscale image (mode='L')
```

```
    """
```

Listing 14.10: ImageSubscriber API

Usage in GUI:

```python
# In gen_gui.py CameraSubscriber (internal class)
class CameraSubscriber(Node):
    image_updated = pyqtSignal(object, bool)  # (frame, is_gray)

    def gray_callback(self, msg):
        cv_image = self.bridge.imgmsg_to_cv2(msg, 'mono8')
        self.image_updated.emit(cv_image, True)

# Connect to GUI update
camera_sub.image_updated.connect(self.update_camera_display)
```

Listing 14.11: Integrating ImageSubscriber with PyQt5

14.11 Bridge Classes

All bridge classes inherit from `ArduinoBridgeBase` and run on the Jetson container.

14.11.1 MotorBridge

```python
class MotorBridge(ArduinoBridgeBase):
    """
    Motor control bridge with multiple input sources.

    Subscriptions (any can control motors):
        - cmd_vel (Twist): Standard ROS teleop messages
        - motor_control_input (String): GUI 6-value format
        - drive_arduino_commands (String): Legacy format

    Publisher:
        - drive_data (String): Arduino response data

    Serial Port: /dev/ttyACM1 (hardcoded)
    """
```

Listing 14.12: MotorBridge Subscriptions

14.11.2 Other Bridges

Table 14.1: Sensor Bridge Summary

Class	Topic	Default Port	Data Format
GPSBridge	gps_data	/dev/ttyACM0	NMEA sentences
IMUBridge	imu_data	Auto-detect	"X: \tY: \tZ:"
UltrasonicBridge	ultrasonic_data	/dev/ttyACM0	"d1, d2, d3, "

Appendix A

Environment Variables

Table A.1: Environment Variable Reference

Variable	Default	Description
ROS_DOMAIN_ID	0	ROS 2 domain for multi-device networking
ROS_LOCALHOST_ONLY	0	Allow network communication (0=yes, 1=no)
DISPLAY	:0	X11 display for GUI
QT_X11_NO_MITSHM	1	Qt X11 compatibility
NVIDIA_VISIBLE_DEVICES	all	GPU access (Jetson)
NVIDIA_DRIVER_CAPABILITIES	all	GPU capabilities (Jetson)

Appendix B

Quick Command Reference

```
# Docker
docker build -t urc_jetson .
docker-compose up -d
docker exec -it pitt_urc_jetson bash
docker logs pitt_urc_jetson

# ROS 2
source /opt/ros/humble/local_setup.bash
ros2 node list
ros2 topic list
ros2 topic echo /topic_name
ros2 topic pub --once /topic std_msgs/String "data: 'test'"

# GUI
python3 -m guis.gen_gui
python3 -m guis.arduino_gui
python3 -m guis.json_motorGUI

# Supervisor (inside Jetson container)
supervisorctl status
supervisorctl restart motor_bridge
supervisorctl tail -f motor_bridge

# Serial
ls /dev/ttyACM*
screen /dev/ttyACM0 115200
```

Listing B.1: Quick Command Reference

Appendix C

Network Ports

Table C.1: Network Port Reference

Port	Protocol	Service
7400	UDP	ROS 2 DDS Discovery
8000	HTTP	Motor/Arm GUI server
11311	TCP	ROS Master (legacy)

Document Information

Field	Value
Document Version	2.0
Last Updated	January 2026
Maintained by	University of Pittsburgh Robotics Club
Repository	github.com/pitt-robotics/URC

For questions or contributions, please refer to `CONTRIBUTING.md` or open an issue on GitHub.