**GPS Waypoint-Based Autonomous Navigation (ROS)  
Repository: ArghyaChatterjee/gps-waypoint-based-autonomous-navigation-in-ros**

**Overview**  
A ROS package that drives a rover through GPS waypoints while avoiding obstacles, using RTK-GPS, IMU, wheel odometry and LiDAR

**Code Layout  
gps\_rover\_navigation/**

**│**

**├── config/**

**│ └── waypoints.txt # List of GPS lat/lon**

**│**

**├── utils/**

**│ ├── gps\_utils.py # Distance, bearing, GPS conversion**

**│ ├── fail\_safes.py # Fail-safe handlers**

**│ └── obstacle\_avoidance.py # Lidar-based avoidance logic**

**│**

**├── core/**

**│ ├── connection.py # Connects to vehicle**

**│ ├── navigator.py # Main waypoint navigation logic**

**│ ├── velocity\_control.py # Local/global NED movement**

**│ └── mission\_manager.py # Mission state machine**

**│**

**├── logs/**

**│ └── gps\_log.csv # Logs of location and status**

**│**

**├── main.py # Main entrypoint**

**└── calibrate\_heading.py # Heading correction**

**STRUCTURE**

**Sensor Fusion**: Utilizes ekf\_localization to fuse odometry data with IMU and GPS inputs.

**Coordinate Transformation:** Employs navsat\_transform to convert GPS data to the robot's odometry coordinate system.

**Mapping and Navigation:** Uses GMapping for map creation and move\_base for goal navigation while avoiding obstacles.

**Custom Nodes:**

**gps\_waypoint:** Reads waypoint files and sends goals to move\_base.

**gps\_waypoint\_continuous1 & gps\_waypoint\_continuous2**: Enable continuous navigation between waypoints.

**collect\_gps\_waypoint:** Allows manual collection of waypoints via joystick.

calibrate\_heading: Sets the robot's heading at startup to address magnetometer inaccuracies.

**plot\_gps\_waypoints:** Saves raw GPS data for plotting.

**gps\_waypoint\_mapping:** Integrates with Mandala Robotics' 3D mapping software.​

**Fail-Safe Features:**  
GPS Anomaly Detector & Auto-Reset: if the rover moves > 0.5 m in 3 s but GPS hasn’t changed, it pauses, restarts the GPS driver (or power-cycles), switches to dead-reckoning until fix returns, and logs a warning  
Emergency Stops  
• **Big Red Button** – physical switch cuts motor power instantly  
• **Software Watchdog** – node halts rover if GPS or IMU stops updating for 2 s  
• **Brake Override** – spring or solenoid brakes engage on power loss

**Heartbeats & Health Checks:**  
• Each key node publishes an “I’m alive” ping every second  
• Missing two pings triggers emergency stop  
• Small display shows GPS quality and battery level; logs detailed health data for review  
**Return-to-Home (RTH)**: if the remote link drops for > 10 s, the rover drives back to its launch GPS point  
**Obstacle Detection:** LiDAR or ultrasonic detects anything within 0.5 m → immediate halt  
**Low-Battery Shutdown:** battery < 20 % → stop and alert operator

**Tilt / Rollover Sensor**  
An onboard inclinometer (IMU) watches chassis pitch/roll. If tilt exceeds, say, 20°, the rover halts to avoid tipping over.

**Motor Overcurrent Protection**  
Inline current sensors on each drive motor. If current spikes beyond a preset limit (e.g. jam or stall), the controller cuts power to that motor.

**GPS Spoofing / Anomaly Alarm**  
Monitor signal strength, sudden large jumps in position, or inconsistent satellite IDs. On detection, log the event, switch to dead-reckoning, and alert the operator.

**Secure & Authenticated Links**  
Encrypt NTRIP and ROS communications (TLS, SSH tunnels or VPN) to prevent man-in-the-middle attacks or data tampering.

**Time-Sync Verification**  
Ensure all sensor timestamps stay within a tight window (e.g. <50 ms drift). If the system clock drifts too far, pause navigation until NTP/GPS time is re-synchronized.

**Environmental Condition Monitor**  
Add simple rain, wind or ambient-light sensors. If conditions exceed safe thresholds (heavy rain, high wind), the rover pauses operations and alerts you.

**Remote E-Stop & Beacon**  
A handheld radio or WebUI button that sends a ROS topic (/remote\_estop). Also broadcast a flashing LED beacon and buzzer so you can visually/audibly find and stop it.

**Thermal / Overheat Protection**  
Watch CPU, GPU and motor-driver temperatures. If any component exceeds safe temp (e.g. 80 °C), throttle performance or halt until it cools.

**Automated Firmware Fallback**  
On boot, verify checksum of your main control firmware. If corrupt or missing, automatically revert to a known-good backup image and continue.

**Redundancy Mechanisms:**

* **Dual GNSS Constellation**s (GPS + GLONASS/Galileo) for uninterrupted fixes  
  Multi-Sensor Fusion (GPS, IMU, odometry, LiDAR—and optionally stereo-vision for canopy) so loss of one sensor is covered by others  
  **Dual Onboard Computers** (Jetson TX2 and Raspberry Pi) running identical ROS stacks; if one crashes, the other continues  
  **Dual RTK Link**s (Wi-Fi and LoRa/LTE) with whichever correction stream is freshest being used  
  **CAN-Bus** Supervisor hardware that resets the bus if error rates exceed safe thresholds

**How to Implement (Plain Steps):**

* **Add GPS Anomaly Watchdog** – in utils/fail\_safes.py subscribe to /odom and /rtk\_fix; if motion > 0.5 m but GPS delta < 0.1 m for 3 s, publish zero cmd\_vel and restart your GPS node
* **Wire Emergency-Stop Buttons** – mount multiple big-red switches in series to a safety relay wired into your motor controller’s kill input
* **Configure Software Watchdog** – use the software\_watchdogs ROS package to monitor critical topics (/rtk\_fix, /imu/data); on timeout automatically trigger a stop callback
* **Set Up Heartbeats** – in every critical node publish a std\_msgs/Header “heartbeat” at 1 Hz; run a heartbeat\_monitor node that stops the rover if two consecutive heartbeats are missed
* **Enable Return-to-Home** – on launch save the initial GPS fix; in your comms-failure handler send that fix as a move\_base goal via actionlib
* **Configure Sensor Fusion** – install robot\_localization and navsat\_transform\_node; list /rtk\_fix, /imu/data, /odom (and /vo if used) as inputs in your EKF YAML
* **Deploy Dual-Compute & Dual-RTK** – clone your workspace to both SBCs and launch under different namespaces; run two RTK serial input nodes and in your fixer logic always pick the most recent valid packet
* **n utils/fail\_safes.py subscribe to /imu/data**. Convert the incoming quaternion to roll/pitch (using tf.transformations.euler\_from\_quaternion). If either angle exceeds 20°, publish a zero cmd\_vel and log a warning.
* **Wire current sensor**s on each drive motor to publish readings on /motor\_current. In utils/fail\_safes.py subscribe to that topic; if any channel stays above your preset limit (e.g. 10 A) for more than 0.5 s, call your emergency stop routine.
* **In your GPS nod**e (e.g. core/connection.py) keep track of the last good fix. On each new fix compare it: if it jumps more than 10 m in one second or the satellite list changes wildly, switch off GPS in your EKF (use dead-reckoning), log the event, and alert the operator.
* **Generate TLS certificates** for each ROS node (using SROS or SROS2 tools). In your launch files add the secure parameters pointing to the cert/key files so that all ROS topics and services run over encrypted, authenticated channels.
* **Install and configure chrony (or ntpd)** on each SBC to sync system time to GPS or NTP servers. In a small health-monitor node compare rospy.Time.now() to system time; if they drift more than 50 ms, pause navigation until they re-sync.
* **Hook up simple rain and wind sensors to a microcontroller** that publishes /rain\_rate and /wind\_speed. In utils/fail\_safes.py subscribe to those topics, and if rain exceeds 2 mm/h or wind exceeds 15 m/s, call your emergency stop and display an alert.
* **In main.py subscribe to a /remote\_estop topic**. When a message arrives, publish zero cmd\_vel, and also toggle a GPIO-driven LED and buzzer so you can see/hear that the rover has been remotely stopped.
* **In utils/health\_monitor.py read CPU/GPU temperature**s from /sys/class/thermal (or via /diagnostics). If any temperature exceeds 80 °C, reduce your maximum speed in velocity\_control.py or call an emergency stop until things cool down.
* **Add a small startup scrip**t (in your launch or systemd unit) that runs an MD5 checksum on your main ROS package folder. If the checksum fails, automatically extract a known-good backup tarball, reinstall it, and then start your ROS nodes.