🚀 Overview

This ROS-based package enables autonomous outdoor navigation using GPS waypoints, while building maps and avoiding obstacles. It is designed for rovers equipped with IMU, Novatel GPS, and Sick lms111 LiDAR.

🧩 Core Components

🔹 Standard ROS Packages:

ekf\_localization: Fuses GPS, IMU, and odometry for robust localization.

navsat\_transform: Converts GPS coordinates to the robot's odometry frame.

GMapping: SLAM-based mapping and obstacle detection.

move\_base: Path planning and goal navigation, with dynamic obstacle avoidance.

🔹 Custom Nodes:

gps\_waypoint: Reads GPS waypoint files, transforms them, and sends navigation goals.

gps\_waypoint\_continuous1/2: Enables continuous waypoint navigation with different controller strategies.

collect\_gps\_waypoint: Lets users manually drive and record waypoints.

calibrate\_heading: Corrects heading at startup to address IMU/magnetometer errors.

plot\_gps\_waypoints: Logs GPS data for visualization/debugging.

gps\_waypoint\_mapping: Integrates with 3D mapping for enhanced autonomous exploration.

🛡️ Redundancy & Safety Features

Sensor Fusion: Maintains localization even if one sensor fails.

Obstacle Avoidance: Dynamic replanning through move\_base.

Fail-safes:

Connection retries and timeouts.

Goal reachability checks to prevent getting stuck.

Transformation time limits for responsiveness.

🏗️ Project Structure

Modular ROS setup with a “Navigation” package.

Launch and config files for integration.

Mostly C++ (72%) and Python (25%) with CMake and shell scripts for build automation.

✅ Conclusion

This system is a well-structured, robust GPS-based autonomous navigation solution in ROS, combining standard tools with custom enhancements for reliable outdoor robotics.

### 🚀 ****Overview****

This ROS-based package enables autonomous outdoor navigation using GPS waypoints, while building maps and avoiding obstacles. It is designed for rovers equipped with **IMU, Novatel GPS, and Sick lms111 LiDAR**.

### 🧩 ****Core Components****

#### 🔹 ****Standard ROS Packages:****

**ekf\_localization:** Fuses GPS, IMU, and odometry for robust localization.

**navsat\_transform:** Converts GPS coordinates to the robot's odometry frame.

**GMapping:** SLAM-based mapping and obstacle detection.

**move\_base:** Path planning and goal navigation, with dynamic obstacle avoidance.

#### 🔹 ****Custom Nodes:****

**gps\_waypoint:** Reads GPS waypoint files, transforms them, and sends navigation goals.

**gps\_waypoint\_continuous1/2:** Enables continuous waypoint navigation with different controller strategies.

**collect\_gps\_waypoint:** Lets users manually drive and record waypoints.

**calibrate\_heading:** Corrects heading at startup to address IMU/magnetometer errors.

**plot\_gps\_waypoints:** Logs GPS data for visualization/debugging.

**gps\_waypoint\_mapping:** Integrates with 3D mapping for enhanced autonomous exploration.

### 🛡️ ****Redundancy & Safety Features****

**Sensor Fusion:** Maintains localization even if one sensor fails.

**Obstacle Avoidance:** Dynamic replanning through move\_base.

**Fail-safes:**

Connection retries and timeouts.

Goal reachability checks to prevent getting stuck.

Transformation time limits for responsiveness.

### 🏗️ ****Project Structure****

Modular ROS setup with a “Navigation” package.

Launch and config files for integration.

Mostly **C++ (72%)** and **Python (25%)** with CMake and shell scripts for build automation.

### ✅ ****Conclusion****

This system is a well-structured, robust GPS-based autonomous navigation solution in ROS, combining standard tools with custom enhancements for reliable outdoor robotics.

Redundancies ensure reliability by providing backup mechanisms and alternative pathways:

* **Sensor Fusion**: ekf\_localization fuses data from odometry, IMU, and GPS, creating a robust state estimate. This redundancy means that if GPS signal is lost (e.g., in urban canyons), the system can rely on IMU and odometry for short-term localization, reducing the risk of failure. This is particularly important for outdoor navigation where GPS reliability can vary.
* **Navigation Stack Redundancies**: move\_base includes path planning redundancies, with the global planner finding alternative routes if the initial path is blocked, and the local planner (teb\_local\_planner) adjusting for dynamic obstacles. This ensures the robot can navigate around unexpected blockages, enhancing operational continuity.
* **Waypoint File Redundancy**: Storing waypoints in a file (points\_outdoor.txt) allows for easy updates without code changes, providing flexibility. If the robot fails to reach a waypoint, the file can be edited to adjust the mission, offering a backup planning mechanism.
* **Modular ROS Architecture**: The use of separate nodes allows for individual node restarts or replacements if one fails, adding another layer of redundancy. For example, if gps\_waypoint crashes, it can be relaunched without affecting other nodes like move\_base.

These redundancies contribute to a resilient system, capable of handling sensor failures and environmental challenges, aligning with best practices in autonomous navigation.

#### Fail-Safe Features

Fail-safe features are implemented to prevent unsafe or undesirable states, ensuring the system operates reliably:

* **Server Availability Check**: The gps\_waypoint node attempts to connect to the move\_base server up to 3 times, with a 5.0-second timeout per attempt. This prevents the system from hanging indefinitely if the server is unavailable, ensuring timely error detection and recovery.
* **Goal Reachability Monitoring**: The code checks if each waypoint is reached; if not, it logs an error ("Husky was unable to reach its goal. GPS Waypoint unreachable.") and shuts down, preventing infinite loops or unsafe navigation attempts. This ensures the robot does not continue in an unsafe state.
* **Transformation Timeouts**: Uses a 3.0-second timeout for coordinate transformations (e.g., GPS to UTM, UTM to map frame) using tf::TransformListener, preventing delays if tf data is unavailable. This enhances system responsiveness and prevents stalling.
* **Obstacle Avoidance Integration**: Through move\_base and the local planner, the system dynamically adjusts paths for obstacles, with logging for debugging, ensuring safety in operation. If an obstacle is detected, the robot can stop or replan, avoiding collisions.
* **Parameter Limits in move\_base**: While not explicitly detailed, standard move\_base configurations likely include parameters like maximum velocity and acceleration limits, ensuring safe operation speeds and preventing abrupt movements that could destabilize the robot.

These fail-safe mechanisms ensure the system can handle errors gracefully, maintaining safety and reliability, particularly in outdoor environments where conditions can be unpredictable.

#### Comparison and Context

To provide context, similar repositories were considered, such as [navit-gps/navit]([invalid url, do not cite]) for car navigation and [sieuwe1/Autonomous-Ai-drone-scripts]([invalid url, do not cite]) for drones, but they focus on different aspects (human-driven navigation and person-following, respectively). The fork by ArghyaChatterjee mentions using Mapviz for visualization, but the original repository's focus is on navigation, with RViz likely the default visualization tool, given standard ROS practices.

The system's design aligns with ROS community standards, leveraging packages like those from [Clearpath Robotics]([invalid url, do not cite]) and [nickcharron]([invalid url, do not cite]), acknowledging contributions that enhance reliability and extensibility.

#### Conclusion

The analysis reveals that mission-mangal/GPS-waypoint-based-Autonomous-Navigation-in-ROS employs a modular structure using ROS, with redundancies in sensor fusion and navigation planning, and robust fail-safe features like server checks and timeouts. This implementation is suitable for robotics applications, ensuring reliable and safe navigation in outdoor environments, with a clear separation of concerns and scalability for future enhancements.

GPS-waypoint-based-Autonomous-Navigation-in-ROS

GPS points will be predefined in ROS based robots to navigate to the destination avoiding obstacles. This package performs outdoor GPS waypoint navigation. It can navigate while building a map, avoiding obstacles, and can navigate continuously between each goal or stop at each goal.

This repo is made to run on a Rover with IMU, Novatel GPS, and Sick lms111 lidar.

This package uses a combination of the following packages:

ekf\_localization to fuse odometry data with IMU and GPS data.

navsat\_transform to convert GPS data to odometry and to convert latitude and longitude points to the robot's odometry coordinate system.

GMapping to create a map and detect obstacles.

move\_base to navigate to the goals while avoiding obstacles (goals are set using recorded or inputted waypoints).

The Navigation package within this repo includes the following custom nodes:

gps\_waypoint to read the waypoint file, convert waypoints to points in the map frame and then send the goals to move\_base.

gps\_waypoint\_continuous1 and gps\_waypoint\_continuous2 for continuous navigation between waypoints using two seperate controllers.

collect\_gps\_waypoint to allow the user to drive the robot around and collect their own waypoints.

calibrate\_heading to set the heading of the robot at startup and fix issues with poor magnetometer data.

plot\_gps\_waypoints to save raw data from the GPS for plotting purposes.

gps\_waypoint\_mapping to combine waypoint navigation with Mandala Robotics' 3D mapping software for autonomous 3D mapping.

🛩️ Drone Control Scripts

1. Autonomous Flight Control

Uses DroneKit to connect, arm, and fly the drone to 3 GPS waypoints.

Returns to home using RTL (Return to Launch).

Monitors speed and progress.

2. Velocity and Navigation Control

Controls drone using local and global NED frames.

Performs directional movements: forward, reverse, turn, and compass-based.

Uses MAVLink for velocity commands.

Supports GPS-free backup movement.

3. ArUco Marker-Based Precision Parking

Uses OpenCV and ArUco for vision-based precise parking.

Guides drone/rover to align and park near detected markers.

🚗 Rover Control Scripts

1. Location-Based Movement

Navigates rover using predefined GPS waypoints.

Employs simple\_goto from DroneKit for point-to-point movement.

2. Velocity-Based Movement

Controls rover using NED velocity commands.

Enables precise and smooth maneuverability.

🧰 Templates and Tests

1. DroneKit Control Template

Starter code for both position and velocity control.

Includes connection, arming, and basic movement.

2. DroneKit Connection Test

Tests basic connectivity and arming.

Verifies DroneKit setup and communication.

📦 Common Dependencies

DroneKit, pymavlink, OpenCV, NumPy, imutils, argparse, time, socket

These scripts form a comprehensive toolkit for autonomous drone and rover development, focusing on GPS waypoint navigation, velocity control, computer vision-based precision, and foundational DroneKit connectivity.

Absolutely! If you're building a **GPS-based autonomous navigation system for a rover in a farm** using **Python**, here's a complete, **step-by-step guide** with code structure, redundancy strategies, and fail-safe mechanisms—based on best practices from ROS/DroneKit systems like the ones you've shared.

## 🧭 ****Step-by-Step Plan: GPS-Based Autonomous Rover Navigation****

### ****1. Define the Goal****

Navigate a rover autonomously through **predefined GPS waypoints** (e.g., row-by-row in a farm).

Avoid obstacles.

Return to home if mission fails or completes.

### ****2. Hardware & Software Setup****

**Hardware Required:**

GPS (e.g., Ublox or Novatel RTK-GPS)

IMU

Lidar or ultrasonic sensors (for obstacle detection)

Rover with motor control (via companion computer)

Companion computer (e.g., Raspberry Pi, Jetson Nano)

**Software Stack (Python-based):**

**DroneKit** (for MAVLink communication)

**pymavlink**

**OpenCV** (for vision if needed, e.g., ArUco or row tracking)

**matplotlib** or **plotly** (for visualization)

Optional: **ROS** (for modularity and sensor fusion)

### ****3. Code Structure****

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

### ****4. Redundancies and Fail-Safes****

#### ✅ ****Redundancy Features****

**Sensor Fusion**: Combine GPS, IMU, wheel encoders for location reliability (EKF-like logic if not using ROS).

**Multiple Navigation Modes**: Use velocity\_control.py for fallback if simple\_goto fails.

**Waypoint Retry Logic**: If goal isn’t reached in N seconds, reattempt or skip.

**Manual Override Mode**: Pause/resume mission with keyboard or remote control.

#### 🔒 ****Fail-Safe Features****

**Connection Loss Detection**: Return to launch (RTL) if lost for > T seconds.

**Stuck Detection**: If velocity < threshold for N seconds, trigger avoidance or reroute.

**Obstacle Avoidance**: Stop or reroute using lidar feedback.

**Battery Check (if supported)**: Return home if voltage drops below threshold.

**Timeouts**: Apply timeouts on all waypoint moves to avoid infinite loops.

### ****5. Core Navigation Logic****

In navigator.py:

def goto\_waypoint(vehicle, lat, lon, alt):

point = LocationGlobalRelative(lat, lon, alt)

vehicle.simple\_goto(point)

start = time.time()

while time.time() - start < TIMEOUT:

current = vehicle.location.global\_relative\_frame

dist = get\_distance\_meters(current, point)

if dist < WAYPOINT\_THRESHOLD:

return True

time.sleep(1)

return False # Timeout reached

### ****6. Loop Through All Waypoints****

In mission\_manager.py:

for wp in waypoints:

success = goto\_waypoint(vehicle, wp.lat, wp.lon, wp.alt)

if not success:

log\_error("Waypoint unreachable", wp)

trigger\_failsafe("WAYPOINT\_FAIL")

break

### ****7. Visualization and Debugging****

Plot live GPS track with matplotlib

Log all events in gps\_log.csv

Use OpenCV to visualize row alignment or detect ArUco markers (if you add vision features)

### ✅ Final Tip: Keep It Modular

Write every function to be testable and modular.

Use status publishers/loggers so you can monitor behavior remotely (e.g., via SSH or dashboard).

Want me to scaffold this code structure or create a launchable Python template for your project?