# RTK GPS-Based Rover for Farm Navigation

# Det****ailed Documentation****

## Table of Contents

1. **Introduction**
2. **Background**
3. **Objectives**
4. **System Architecture**
5. **Module Descriptions**  
   5.1 coordinate\_converter.py  
   5.2 farm\_safety.py  
   5.3 emlid\_gps\_integration.py  
   5.4 farm\_simulation.py  
   5.5 gps\_system\_monitor.py  
   5.6 ntrip\_client.py  
   5.7 logging\_100mm.py  
   5.8 rover\_health\_check.py  
   5.9 row\_navigation.py  
   5.10 sleep\_mode.py
6. **Waypoints Configuration**
7. **Setup and Configuration**
8. **Usage**
9. **Testing and Validation**
10. **Results and Discussion**
11. **Conclusion**
12. **References**

# RTK GPS-Based Rover for Farm Applications: Detailed Documentation

## Introduction

This project develops an autonomous rover for farm applications, leveraging Real-Time Kinematic (RTK) GPS technology to achieve centimeter-level positioning accuracy. The rover navigates through farm rows in a zigzag pattern, optimizing tasks such as planting, spraying, or monitoring crops. The system integrates multiple Python modules to manage coordinate conversions, safety checks, GPS data integration, simulation, monitoring, logging, health checks, navigation, and failsafe mechanisms, providing a robust solution for precision agriculture.

## Background

RTK GPS enhances the precision of position data derived from GPS, achieving accuracy up to 1 cm by using real-time correction data from reference stations [ArduSimple](https://www.ardusimple.com/precision-agriculture/). In precision agriculture, RTK GPS is critical for tasks requiring high accuracy, such as autonomous tractor navigation, precision planting, and variable rate applications, which reduce resource waste and improve efficiency [AgriTechTomorrow](https://www.agritechtomorrow.com/news/2023/09/13/rtk-makes-precision-agriculture-simple/14836/). The rover in this project leverages RTK GPS to navigate farm rows with minimal overlap, optimizing operations like seeding and fertilization [SMAJAYU](https://www.smajayu.com/rtk-gps/).

## Objectives

The project aims to:

* Achieve centimeter-level positioning accuracy using RTK GPS.
* Navigate the rover through farm rows in an efficient zigzag pattern.
* Ensure safe operation within defined farm boundaries, avoiding revisiting waypoints.
* Monitor and log performance data for analysis and debugging.
* Provide simulation capabilities for testing without physical hardware.

## System Architecture

The system comprises several Python modules, each handling specific functionalities, as summarized in the following table:

| **Module Name** | **Functionality** |
| --- | --- |
| coordinate\_converter.py | Converts between WGS84 latitude/longitude and UTM coordinates (zone 45N), calculates distances and bearings. |
| farm\_safety.py | Ensures safe navigation by preventing boundary exits and waypoint revisits, manages drift. |
| emlid\_gps\_integration.py | Integrates with Emlid Reach GNSS receiver for accurate positioning data. |
| farm\_simulation.py | Simulates rover navigation with real or simulated GPS data, visualizes movement. |
| gps\_system\_monitor.py | Monitors GPS and NTRIP connectivity. |
| ntrip\_client.py | Receives real-time correction data via NTRIP for enhanced GPS accuracy. |
| logging\_100mm.py | Logs GPS data at 100ms intervals for analysis. |
| rover\_health\_check.py | Performs comprehensive health checks on the GPS system. |
| row\_navigation.py | Manages navigation through farm rows, including zigzag patterns. |
| sleep\_mode.py | Handles failsafe conditions, including entering sleep mode. |

## Module Descriptions

### coordinate\_converter.py

**Purpose**: This module facilitates conversions between WGS84 latitude/longitude (EPSG:4326) and UTM coordinates (EPSG:32645, zone 45N), and computes distances and bearings essential for navigation.

**Key Class**: CoordinateConverter

**Key Methods**:

* \_\_init\_\_(): Initializes with default UTM zone settings (zone 45, letter 'N').
* latlon\_to\_utm\_coord(lat, lon): Converts latitude and longitude to UTM coordinates (easting, northing, zone number, zone letter). Validates easting values (100,000–999,999 meters).
* utm\_to\_latlon\_coord(easting, northing, zone\_number=None, zone\_letter=None): Converts UTM coordinates back to latitude and longitude, defaulting to zone 45N.
* get\_distance\_between\_coords(lat1, lon1, lat2, lon2): Calculates great-circle distance using the haversine formula (Earth's radius: 6,371 km).
* get\_bearing\_between\_coords(lat1, lon1, lat2, lon2): Computes bearing (0–360 degrees) from one point to another.

**Integration**: Used by emlid\_gps\_integration.py and row\_navigation.py for coordinate transformations and spatial calculations.

### farm\_safety.py

**Purpose**: Ensures safe navigation by preventing the rover from exiting farm boundaries or revisiting waypoints, and manages navigation drift.

**Key Class**: SafetyModule

**Key Methods**:

* \_\_init\_\_(failsafe=None, revisit\_threshold=0.2): Initializes with optional failsafe linkage and revisit threshold.
* set\_geofence(vertices): Defines the farm boundary as a polygon.
* set\_waypoints(waypoints): Sets navigation waypoints and clears visited indices.
* is\_outside\_geofence(pos): Uses ray-casting to check if a position is outside the geofence.
* check\_safety(pos, heading, path): Evaluates safety, checking for no-go violations (boundary exits, waypoint revisits) and drift, returning statuses like 'safe', 'no-go', or 'drift'.
* handle\_no\_go\_violation(pos, heading, violation\_data): Manages no-go violations (placeholder).
* handle\_drift(pos, heading, drift\_data): Handles drift correction (placeholder).
* find\_closest\_point\_on\_path(pos, path): Finds the closest point on the path to a given position.
* diff\_h(c, t): Calculates the difference between two headings, ensuring the result is within -180 to 180 degrees.

**Integration**: Interacts with farm\_simulation.py and row\_navigation.py to enforce safety during navigation. Uses FailsafeModule for critical error handling.

### emlid\_gps\_integration.py

**Purpose**: Interfaces with the Emlid Reach GNSS receiver to provide real-time, high-precision positioning data, supporting both real and simulated data.

**Key Class**: EmlidGPSReader

**Key Methods**:

* \_\_init\_\_(port='COM12', baud\_rate=115200, message\_format='nmea'): Initializes with serial port, baud rate, and data format.
* connect(retries=10, retry\_delay=2): Establishes a serial connection with retry logic.
* start\_reading(): Starts a background thread to read GPS data at 10Hz.
* register\_callback(callback\_func): Registers a callback for new GPS data.
* get\_last\_position(): Returns the most recent GPS position data.
* check\_health(): Returns system health status (connection, fix quality, satellite count).
* \_read\_nmea(): Parses NMEA sentences (e.g., GPGGA, GPRMC).
* \_read\_json(): Parses JSON data from the receiver.
* \_simulate\_gps\_data(): Generates simulated GPS data for testing.

**Integration**: Provides GPS data to farm\_simulation.py and logging\_100mm.py, converting coordinates using coordinate\_converter.py.

### farm\_simulation.py

**Purpose**: Orchestrates the simulation of rover navigation, integrating all modules and visualizing movement using Matplotlib.

**Key Components**:

* **Rover**: Manages rover state (position, heading, UTM coordinates).
* **RowNavigator**: Plans and executes navigation paths.
* **SafetyModule**: Enforces safety constraints.
* **FailsafeModule**: Handles critical errors.
* **RoverHealthCheck**: Performs pre-simulation health checks.
* **EmlidGPSReader**: Supplies GPS data.
* **CoordinateConverter**: Handles coordinate transformations.
* **NTRIPClient**: Manages RTK corrections.
* **GPSSystemMonitor**: Monitors GPS health.

**Key Functions**:

* run\_simulation(): Initializes the simulation, performs health checks, sets up GPS and NTRIP, plans the path, and visualizes progress.
* navigate\_to\_path\_start(rover, safety, path\_start, ax, fig, rover\_patch): Navigates to the path's starting point.
* visualize\_turn(rover, desired\_heading, ax, fig, rover\_patch): Visualizes heading alignment.
* update\_rover\_visualization(rover, ax, fig, rover\_patch): Updates the Matplotlib plot with the rover's position.

**Workflow**:

1. Initialize logging and error handling.
2. Perform health checks using rover\_health\_check.py.
3. Set up GPS with optional NTRIP corrections.
4. Load or generate waypoints using row\_navigation.py.
5. Navigate to the path start, align heading, and follow the path.
6. Visualize the farm, path, and rover movement.
7. Clean up resources upon completion or error.

**Integration**: Central module that coordinates all others, ensuring seamless operation.

### gps\_system\_monitor.py

**Purpose**: Monitors GPS and NTRIP connectivity to ensure continuous operation.

**Key Class**: GPSSystemMonitor

**Key Methods**:

* \_\_init\_\_(emlid\_reader, ntrip\_client=None): Initializes with Emlid reader and optional NTRIP client.
* start\_monitoring(): Starts the monitoring thread.
* stop\_monitoring(): Stops the monitoring thread.
* \_monitor\_loop(): Continuously checks GPS and NTRIP status, reconnecting if necessary.
* cleanup(): Releases resources.

**Integration**: Works with emlid\_gps\_integration.py and ntrip\_client.py to maintain reliable connections.

### ntrip\_client.py

**Purpose**: Receives real-time correction data via NTRIP for RTK accuracy.

**Key Class**: NTRIPClient

**Key Methods**:

* \_\_init\_\_(host, port, mountpoint, user, password): Initializes with NTRIP server details.
* connect(max\_retries=3, retry\_delay=2): Establishes a connection with retry logic.
* get\_corrections(): Streams correction data.
* check\_connection(): Verifies connection status.
* cleanup(): Closes the session.

**Integration**: Provides corrections to emlid\_gps\_integration.py for enhanced accuracy.

### logging\_100mm.py

**Purpose**: Logs GPS data at 100ms intervals to a CSV file for analysis.

**Key Class**: GPSLogger

**Key Methods**:

* \_\_init\_\_(rover): Initializes with a rover object, sets log file path (F:\GPS\task\_2\_waypoints\rover\_log.csv).
* start(): Starts the logging thread.
* stop(): Stops the logging thread.
* log\_data\_once(): Logs a single GPS data entry.
* get\_gps\_data(): Collects current GPS data, including position and fix quality.
* process\_emlid\_gps\_data(emlid\_data): Processes real GPS data, converting to UTM.

**Integration**: Uses data from emlid\_gps\_integration.py and coordinate\_converter.py, logs to a CSV file.

### rover\_health\_check.py

**Purpose**: Performs comprehensive health checks on the GPS system to ensure reliability.

**Key Classes**:

* RTKGPSRover: Handles GPS data reading.
* RoverHealthCheck: Performs health checks.

**Key Methods**:

* check\_rtk\_status(): Verifies RTK fix status.
* check\_satellite\_count(): Ensures sufficient satellites.
* check\_dop\_values(): Validates DOP values.
* check\_position\_validity(): Checks position within boundaries.
* run\_all\_checks(continue\_on\_failure=True, simulation\_mode=False): Executes all checks.
* generate\_health\_report(): Summarizes check results.

**Integration**: Used by farm\_simulation.py to verify system readiness.

### row\_navigation.py

**Purpose**: Manages navigation through farm rows, generating and following zigzag patterns.

**Key Class**: RowNavigator

**Key Methods**:

* load\_rows\_from\_csv(csv\_filename): Loads waypoints from a CSV file, normalizing coordinates.
* generate\_rows(start\_x, start\_y, num\_strips): Creates a zigzag path with specified spacing.
* move\_precisely\_to\_point(target\_point): Moves the rover to a waypoint with precision.
* navigate\_path(): Executes the full path, handling row transitions.
* calculate\_deviation(x, y): Measures deviation from the planned path.

**Integration**: Works with farm\_simulation.py and farm\_safety.py to navigate waypoints.

### sleep\_mode.py

**Purpose**: Handles failsafe conditions, including entering sleep mode to prevent unsafe operation.

**Key Class**: Likely FailsafeModule (based on related functionality).

**Key Methods** (Inferred):

* Monitor for anomalies (e.g., stale GPS data, position jumps).
* Attempt recovery or enter sleep mode if issues persist.

**Integration**: Interacts with farm\_safety.py and farm\_simulation.py to manage critical errors.

## Waypoints Configuration

The waypoints must be provided in a CSV file with the following format:

id,left,top,right,bottom,row\_index,col\_index,id\_2,distance,angle,x,y,start\_point

Example:

1,380000,2044900,380100,2044900,0,0,1,0,90.000000000000,380000.000000000,2044900,1

1,380000,2044900,380100,2044900,0,0,1,10,90.000000000000,380010.000000000,2044900,0

1,380000,2044900,380100,2044900,0,0,1,20,90.000000000000,380020.000000000,2044900,0

1,380000,2044900,380100,2044900,0,0,1,30,90.000000000000,380030.000000000,2044900,0

1,380000,2044900,380100,2044900,0,0,1,40,90.000000000000,380040.000000000,2044900,0

1,380000,2044900,380100,2044900,0,0,1,50,90.000000000000,380050.000000000,2044900,0

1,380000,2044900,380100,2044900,0,0,1,60,90.000000000000,380060.000000000,2044900,0

Each row includes:

* **id**: Waypoint or row identifier.
* **left, top, right, bottom**: Coordinates defining the bounding box of the row or area.
* **row\_index, col\_index**: Indices indicating the row and column in the farm grid.
* **id\_2**: Additional identifier for grouping or sequencing.
* **distance**: Distance from a reference point or along the path (in meters).
* **angle**: Bearing or angle at the waypoint (in degrees).
* **x, y**: UTM coordinates (easting and northing).
* **start\_point**: Flag (1 or 0) indicating if this is the starting point of a row or path.

Ensure the CSV file is correctly formatted and located in the specified path (e.g., F:\GPS\task\_2\_waypoints\waypoints\_100mm.csv) for the simulation to load it properly.

## Setup and Configuration

**Hardware Setup**:

* + Connect the Emlid Reach GNSS receiver to the rover.
  + Configure the receiver for RTK operation, ensuring proper antenna placement.
  + Verify power supply and communication interfaces.

**Software Dependencies**:

* + Install Python 3.x and required libraries: serial, requests, matplotlib, utm, numpy, pandas.
  + Ensure compatibility by using the latest stable versions of these libraries.

**Configuration Parameters**:

* + **NTRIP Settings**: Specify host, port, mountpoint, username, and password for the NTRIP server.
  + **GPS Settings**: Configure serial port (e.g., COM12) and baud rate (e.g., 115200).
  + **Waypoints**: Provide the waypoints CSV file in the specified format.
  + **Geofence**: Define farm boundary coordinates as a polygon for safety checks.

## Usage

**Running the Simulation**:

* + Execute farm\_simulation.py to start the simulation.
  + The script performs health checks, sets up GPS and NTRIP, loads waypoints, and begins navigation.
  + Monitor the Matplotlib visualization for real-time rover movement and path progress.

**Interpreting Outputs**:

* + **Logs**: Stored in F:\GPS\task\_2\_waypoints\rover\_log.csv, containing timestamped data on position, heading, fix quality, and more.
  + **Visualizations**: Matplotlib plots display farm boundaries, waypoints, and the rover’s trajectory.
  + **Health Reports**: Generated by rover\_health\_check.py, saved in rover\_health.csv.

## Testing and Validation

* **Simulation Mode**: Uses simulated GPS data to test navigation algorithms and safety protocols without hardware, ideal for development and debugging.
* **Real-World Testing**: Deploy the rover in a farm environment to verify accurate row navigation, minimal overlap, and resource efficiency, aligning with RTK GPS benefits [SMAJAYU](https://www.smajayu.com/rtk-gps/).

## Results and Discussion

The rover is designed to navigate farm rows with high precision, reducing overlap and optimizing resource use, consistent with RTK GPS benefits in agriculture [ArduSimple](https://www.ardusimple.com/precision-agriculture/). Challenges like GPS signal loss or drift are mitigated by robust failsafe mechanisms, including automatic reconnection and sleep mode activation. The system’s logging and visualization capabilities facilitate debugging and performance analysis, ensuring reliable operation.

## Conclusion

The RTK GPS-based rover demonstrates the effective application of high-precision positioning in autonomous farm navigation. Future enhancements could include integrating additional sensors (e.g., LiDAR for obstacle detection), optimizing path planning algorithms for complex farm layouts, or expanding simulation capabilities for diverse scenarios.