1. farm\_simulation.py

import matplotlib.pyplot as plt

import time

import numpy as np

import random

import logging\_100mm

# Import our modules

from row\_navigation import Rover, navigate\_to\_point, TOLERANCE, follow\_path\_precisely, update\_rover\_visualization, visualize\_turn

from row\_navigation import RowNavigator

from farm\_safety import SafetyModule

from sleep\_mode import FailsafeModule, GPSFailsafeReason, DriftSeverity, DriftAction

# Import the health check module from the second file

from rover\_health\_check import RoverHealthCheck, HealthCheckFailure

# Import coordinate converter

from coordinate\_converter import CoordinateConverter

import threading

from ntrip\_client import NTRIPClient

from emlid\_gps\_integration import EmlidGPSReader

debug = False

safety = SafetyModule()

def get\_float(prompt):

"""Get a float value from user with error handling"""

while True:

try:

value = float(input(prompt))

return value

except ValueError:

print("⚠️ Please enter a valid number.")

def random\_position\_in\_farm(min\_x, max\_x, min\_y, max\_y, safety\_margin=2.0):

"""Generate a random position inside the farm with a safety margin from boundaries"""

x = random.uniform(min\_x + safety\_margin, max\_x - safety\_margin)

y = random.uniform(min\_y + safety\_margin, max\_y - safety\_margin)

return x, y

def safe\_remove(element):

if element:

try:

element.remove()

return True

except:

if debug: print(f"Warning: failed to remove {element}")

return False

def process\_emlid\_gps\_data(rover, emlid\_data):

"""

Process GPS data from Emlid receiver and update rover position.

This is a standalone version that doesn't rely on class methods.

Args:

rover: The rover instance

emlid\_data: Dictionary with GPS data from Emlid receiver

Returns:

bool: True if position was updated successfully, False otherwise

"""

if not emlid\_data or 'latitude' not in emlid\_data or 'longitude' not in emlid\_data:

print("⚠️ Invalid or missing Emlid GPS data")

return False

try:

# Get the coordinate converter

converter = None

if hasattr(rover, 'coordinate\_converter'):

converter = rover.coordinate\_converter

elif hasattr(rover, 'gps\_logger') and hasattr(rover.gps\_logger, 'converter'):

converter = rover.gps\_logger.converter

else:

print("⚠️ No coordinate converter found")

return False

# Convert lat/lon to UTM

easting, northing = converter.latlon\_to\_utm\_coord(

emlid\_data['latitude'],

emlid\_data['longitude']

)

if easting is None or northing is None:

print("⚠️ Failed to convert lat/lon to UTM")

return False

# Get the correct UTM offsets

utm\_offset\_x = 0

utm\_offset\_y = 0

if hasattr(rover, 'navigator') and hasattr(rover.navigator, 'utm\_offset\_x'):

utm\_offset\_x = rover.navigator.utm\_offset\_x

utm\_offset\_y = rover.navigator.utm\_offset\_y

# Calculate the local coordinates by removing the offsets

local\_x = easting - utm\_offset\_x

local\_y = northing - utm\_offset\_y

# Update rover position

rover.set\_position(local\_x, local\_y)

# Log the update if a logger is available

if hasattr(rover, 'gps\_logger'):

if hasattr(rover.gps\_logger, 'log\_data\_once'):

rover.gps\_logger.log\_data\_once()

return True

except Exception as e:

print(f"Error processing Emlid GPS data: {e}")

return False

def run\_simulation():

def on\_failsafe\_triggered(reason):

print(f"⚠️ Failsafe triggered: {reason.value}")

rover.log\_movement("stop") # Stop the rover for safety

def on\_recovery\_attempt(reason):

print(f"🔄 Attempting recovery from {reason.value}")

current\_time = time.time()

if reason == GPSFailsafeReason.GPS\_STALE\_DATA or reason == GPSFailsafeReason.GPS\_DATA\_LOSS:

failsafe.last\_gps\_update = current\_time

elif reason == GPSFailsafeReason.INTERNET\_CONNECTION\_LOST or reason == GPSFailsafeReason.INTERNET\_CONNECTION\_SLOW:

failsafe.last\_internet\_check = current\_time

elif reason == GPSFailsafeReason.MODULE\_COMMUNICATION\_FAILURE:

failsafe.last\_module\_comm = current\_time

return True # Assume recovery succeeds for simulation

print("🚜 Farm Rover Navigation Simulation 🚜")

print("=====================================")

# -------------------- HEALTH CHECK SECTION --------------------

print("\n🔍 Running rover health checks before simulation...")

# Create the rover first (needed by the health checker)

rover = Rover()

# Initialize the coordinate converter

coordinate\_converter = CoordinateConverter()

rover.coordinate\_converter = coordinate\_converter

# Initialize health checker with the rover instance

health\_checker = RoverHealthCheck(rover)

try:

# Run all health checks

health\_status = health\_checker.run\_all\_checks(simulation\_mode=True)

# Generate and display health report

health\_report = health\_checker.generate\_health\_report()

print(health\_report)

# Check if all systems passed

if not all(health\_status.values()):

print("\n⚠️ One or more health checks failed. Aborting simulation.")

print(" Please address the issues and try again.")

return

print("\n✅ All health checks passed! Proceeding with simulation.")

except HealthCheckFailure as e:

print(f"\n❌ Critical health check failure: {e}")

print(" Simulation cannot proceed until this issue is resolved.")

return

# -------------------- END HEALTH CHECK SECTION --------------------

plt.rcParams['figure.max\_open\_warning'] = 50

failsafe = FailsafeModule()

safety = SafetyModule(failsafe=failsafe)

failsafe.set\_safety\_module(safety)

# Initialize failsafe first

rover.failsafe = failsafe

failsafe.update\_gps\_status(has\_fix=True, satellites=10, hdop=1.0)

failsafe.update\_internet\_status(connected=True, latency=0.1)

failsafe.update\_module\_communication()

failsafe.set\_callbacks(on\_failsafe\_triggered, on\_recovery\_attempt)

# Now initialize GPS logger

gps\_logger = logging\_100mm.initialize\_gps\_logger(rover)

# Initialize GPS reader for NMEA and corrections

emlid\_reader = EmlidGPSReader(message\_format='nmea')

if not emlid\_reader.connect(retries=5, retry\_delay=3):

print("❌ Failed to connect to Emlid receiver")

return

emlid\_reader.start\_reading()

rover.gps\_reader = emlid\_reader

NTRIP\_CONFIG = {

'host': 'your.ntrip.server.url',

'port': 2101,

'mountpoint': 'MOUNTPOINT',

'user': 'your\_username',

'password': 'your\_password'

}

ntrip\_client = NTRIPClient(\*\*NTRIP\_CONFIG)

# Start NTRIP corrections in a background thread

def stream\_rtcm\_corrections():

for rtcm\_data in ntrip\_client.get\_corrections():

if hasattr(rover, 'gps\_reader'): # Ensure EmlidGPSReader is attached

rover.gps\_reader.send\_rtcm\_data(rtcm\_data)

ntrip\_thread = threading.Thread(target=stream\_rtcm\_corrections, daemon=True)

ntrip\_thread.start()

# === END ADDITION ===

# Create row navigator

navigator = RowNavigator(rover)

rover.navigator = navigator

# Start failsafe monitoring

failsafe.start\_monitoring()

navigator.zigzag\_pattern = True

# Load waypoints from CSV file

csv\_loaded = navigator.load\_rows\_from\_csv(r"F:\GPS\task\_2\_waypoints\waypoints\_100mm.csv")

if not csv\_loaded:

print("❌ Failed to load waypoints from CSV. Simulation cannot proceed without waypoints.")

return

# Calculate farm boundaries based on waypoints with margin

margin = 3.0 # Add margin around waypoints

min\_x = min(point[0] for point in navigator.interpolated\_path) - margin

max\_x = max(point[0] for point in navigator.interpolated\_path) + margin

min\_y = min(point[1] for point in navigator.interpolated\_path) - margin

max\_y = max(point[1] for point in navigator.interpolated\_path) + margin

print(f"📏 Dynamic farm boundaries: X [{min\_x:.2f}, {max\_x:.2f}], Y [{min\_y:.2f}, {max\_y:.2f}]")

# Create vertices for the farm boundary

verts = [(min\_x, min\_y), (max\_x, min\_y), (max\_x, max\_y), (min\_x, max\_y)]

# Generate a random entry point

side = random.randint(0, 3)

if side == 0: # Bottom side

entry\_x = random.uniform(min\_x, max\_x)

entry\_y = min\_y

elif side == 1: # Right side

entry\_x = max\_x

entry\_y = random.uniform(min\_y, max\_y)

elif side == 2: # Top side

entry\_x = random.uniform(min\_x, max\_x)

entry\_y = max\_y

else: # Left side

entry\_x = min\_x

entry\_y = random.uniform(min\_y, max\_y)

entry\_point = (entry\_x, entry\_y)

# Set geofence in rover and safety module

rover.set\_geofence(verts, entry\_point)

safety.set\_geofence(verts)

# Generate random starting position inside the farm

random\_x, random\_y = random\_position\_in\_farm(min\_x, max\_x, min\_y, max\_y)

print(f"🎲 Randomly placing rover inside farm at: ({random\_x:.3f}, {random\_y:.3f})")

# Initialize visualization

plt.ion()

fig, ax = plt.subplots(figsize=(10, 8))

ax.set\_title("Rover Farm Navigation Simulation")

# Draw farm boundary

farm\_polygon = plt.Polygon(np.array(verts), closed=True,

facecolor='lightgreen', edgecolor='darkgreen', alpha=0.3)

ax.add\_patch(farm\_polygon)

# Mark random start position

ax.scatter(random\_x, random\_y, c='green', s=80, label='Start (Inside)')

# Setup rover path visualization

path\_line, = ax.plot([], [], 'b-', alpha=0.5, label='Path')

ax.path\_line = path\_line

ax.legend(loc='upper left')

# Set rover starting position (inside farm)

rover.set\_position(random\_x, random\_y, force=True, add\_to\_history=False)

rover.inside\_fence = True # Force the rover to be considered inside the farm

rover.fence\_locked = True # Lock the rover inside the farm

rover.history.append((rover.x, rover.y))

rover\_patch = update\_rover\_visualization(rover, ax, fig)

print("\n🚜 TASK 1: Determining farm navigation plan with zigzag pattern...\n")

# Ensure zigzag pattern is enabled

navigator.zigzag\_pattern = True

# Use the waypoints previously loaded from CSV

safety.set\_waypoints(navigator.interpolated\_path)

# Determine plot boundaries based on waypoints

if navigator.interpolated\_path:

wp\_min\_x = min(point[0] for point in navigator.interpolated\_path)

wp\_max\_x = max(point[0] for point in navigator.interpolated\_path)

wp\_min\_y = min(point[1] for point in navigator.interpolated\_path)

wp\_max\_y = max(point[1] for point in navigator.interpolated\_path)

# Use the wider range between farm boundaries and waypoints

plot\_min\_x = min(min\_x, wp\_min\_x)

plot\_max\_x = max(max\_x, wp\_max\_x)

plot\_min\_y = min(min\_y, wp\_min\_y)

plot\_max\_y = max(max\_y, wp\_max\_y)

# Add a larger margin

margin = max(plot\_max\_x - plot\_min\_x, plot\_max\_y - plot\_min\_y) \* 0.15

ax.set\_xlim(plot\_min\_x - margin, plot\_max\_x + margin)

ax.set\_ylim(plot\_min\_y - margin, plot\_max\_y + margin)

else:

# Fallback to original farm boundaries

margin = 3

ax.set\_xlim(min\_x - margin, max\_x + margin)

ax.set\_ylim(min\_y - margin, max\_y + margin)

ax.grid(True)

# Visualize zigzag row pattern

x\_coords, y\_coords = zip(\*navigator.interpolated\_path)

ax.plot(x\_coords, y\_coords, 'b-', alpha=0.5, label='Zig-Zag Path')

# Mark start and end points

path\_start = navigator.interpolated\_path[0]

path\_end = navigator.interpolated\_path[-1]

ax.scatter(path\_start[0], path\_start[1], c='orange', s=50, marker='s', label='Path Start')

ax.scatter(path\_end[0], path\_end[1], c='red', s=50, marker='o', label='Path End')

fig.canvas.draw\_idle()

plt.pause(0.5)

# --- TASK 1: Navigate directly to the path start point ---

print("\n🚜 TASK 1: Navigating directly to path start point...\n")

print(f"🎯 Path start point: ({path\_start[0]:.3f}, {path\_start[1]:.3f})")

print(f"📏 Distance to path start: {rover.distance\_to(\*path\_start):.3f}m")

def on\_rover\_wakeup():

print("Rover has woken up! Resuming operations...")

# Do whatever you need when rover wakes up

# Navigate to path start

def navigate\_to\_path\_start(rover, safety, path\_start, ax, fig, rover\_patch):

"""

Navigate rover to the starting point of the path using direct point-to-point moves

with a larger step size and a slightly more generous tolerance to avoid getting stuck.

"""

print("\n🗺️ Navigating directly to starting point...")

reached\_start, rover\_patch = navigate\_to\_point(

rover,

path\_start[0],

path\_start[1],

ax,

fig,

rover\_patch,

step\_size=1.5, # larger increments per move

tolerance=0.8 # accept slightly further from the exact point

)

return reached\_start, rover\_patch

# Use our custom function to navigate to path start

reached\_start, rover\_patch = navigate\_to\_path\_start(rover, safety, path\_start, ax, fig, rover\_patch)

if not reached\_start:

print("\n⚠️ Could not reach path start point after multiple attempts.")

print(" Try adjusting simulation parameters or path positioning.")

return

#Force rover position to exactly match path start

rover.set\_position(path\_start[0], path\_start[1], force=True)

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

# Mark path start reached

ax.scatter(path\_start[0], path\_start[1], c='lime', s=80, marker='\*', label='Start Reached')

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(1)

print("\n✅ TASK 1 COMPLETE: Successfully reached path start point")

print(f" Current position: ({rover.x:.3f}, {rover.y:.3f})")

# --- TASK 2: Align to the path direction ---

print("\n🚜 TASK 2: Aligning rover to path direction...\n")

# Find next waypoint (should be index 1 since we're at index 0)

navigator.current\_waypoint\_index = 0 # Force to start at the beginning of the path

next\_point = navigator.interpolated\_path[1]

desired\_heading = navigator.calculate\_heading((rover.x, rover.y), next\_point)

# Align to the path direction

rover\_patch = visualize\_turn(rover, desired\_heading, ax, fig, rover\_patch)

print(f" Aligned rover to heading: {desired\_heading:.1f}°")

print("\n✅ TASK 2 COMPLETE: Successfully aligned to path direction")

# --- TASK 3: Navigate through the path ---

print("\n🚜 TASK 3: Starting path navigation pattern...\n")

# Start navigation from the beginning of the path

navigator.current\_waypoint\_index = 0

path\_success = navigator.navigate\_path(ax, fig, rover\_patch)

if not path\_success:

print("\n⚠️ Failed to navigate path. Simulation halted.")

return

# Mark completion of path

final\_point = navigator.interpolated\_path[-1]

ax.scatter(final\_point[0], final\_point[1], c='green', s=100, marker='\*', label='Mission Complete')

ax.legend(loc='upper left')

fig.canvas.draw\_idle()

plt.pause(1)

print("\n🎉 TASK 3 COMPLETE: Successfully navigated the path")

print("\n🏁 SIMULATION COMPLETE! 🏁")

print(f" Total commands executed: {rover.command\_count}")

print(f" Final position: ({rover.x:.3f}, {rover.y:.3f})")

# Keep plot open until closed manually

plt.ioff()

plt.show(block=True)

logging\_100mm.stop\_gps\_logger(rover)

failsafe.stop\_monitoring()

def simulate\_emlid\_gps\_reading():

"""

Simulate an Emlid GPS reading for testing purposes.

Returns a dictionary with lat/lon coordinates and RTK status.

"""

# These are example coordinates - in a real implementation,

# you would get these from the Emlid GPS receiver

# Randomly choose a solution status for demonstration

solution\_statuses = ["fixed", "float", "single", "dgps"]

solution\_status = random.choice(solution\_statuses)

# Determine appropriate HDOP based on solution status

if solution\_status == "fixed":

hdop = random.uniform(0.01, 0.2)

elif solution\_status == "float":

hdop = random.uniform(0.2, 0.5)

elif solution\_status == "dgps":

hdop = random.uniform(0.5, 1.0)

else: # single

hdop = random.uniform(1.0, 2.0)

return {

'latitude': 28.6139, # Example latitude

'longitude': 77.2090, # Example longitude

'solution\_status': solution\_status, # RTK solution status from Emlid

'satellites': random.randint(8, 15), # Number of satellites

'hdop': hdop # Horizontal dilution of precision

}

def test\_emlid\_integration():

"""

Test function to verify Emlid GPS integration with the rover system.

"""

print("🧪 Testing Emlid GPS integration...")

# Create rover instance

rover = Rover()

# Initialize coordinate converter

converter = CoordinateConverter()

rover.coordinate\_converter = converter

# Create row navigator (needed for UTM offsets)

navigator = RowNavigator(rover)

rover.navigator = navigator

# Set default UTM offsets for testing

navigator.utm\_offset\_x = 380000.0

navigator.utm\_offset\_y = 2044880.0

# Initialize GPS logger

gps\_logger = logging\_100mm.initialize\_gps\_logger(rover)

# Simulate Emlid GPS reading

emlid\_data = simulate\_emlid\_gps\_reading()

print(f"📡 Simulated Emlid GPS reading: Lat={emlid\_data['latitude']}, Lon={emlid\_data['longitude']}")

# Use the built-in function from the logging\_100mm module

success = logging\_100mm.update\_rover\_position\_from\_emlid(rover, emlid\_data)

# Check the result

if success:

print("✅ Successfully processed Emlid GPS data")

print(f"🚜 Rover position (UTM): X={rover.x:.3f}, Y={rover.y:.3f}")

# Calculate the actual global UTM coordinates

actual\_easting = rover.x + rover.navigator.utm\_offset\_x

actual\_northing = rover.y + rover.navigator.utm\_offset\_y

# Convert back to lat/lon for verification

lat, lon = converter.utm\_to\_latlon\_coord(

actual\_easting, actual\_northing,

zone\_number=45, zone\_letter='N' # Make sure to use the correct zone

)

print(f"🌐 Rover position (Lat/Lon): {lat:.6f}, {lon:.6f}")

# Calculate difference from original coordinates

original\_lat = emlid\_data['latitude']

original\_lon = emlid\_data['longitude']

lat\_diff = abs(lat - original\_lat)

lon\_diff = abs(lon - original\_lon)

print(f"📊 Conversion difference: Lat={lat\_diff:.8f}, Lon={lon\_diff:.8f}")

if lat\_diff < 0.0001 and lon\_diff < 0.0001:

print("✅ Conversion accuracy check passed")

else:

print("❌ Conversion accuracy check failed - differences too large")

else:

print("❌ Failed to process Emlid GPS data")

# Cleanup

logging\_100mm.stop\_gps\_logger(rover)

print("🧪 Test completed")

if \_\_name\_\_ == "\_\_main\_\_":

try:

# Uncomment to test Emlid integration separately

# test\_emlid\_integration()

# Run the main simulation

run\_simulation()

except KeyboardInterrupt:

print("\n\n🛑 Simulation terminated by user.")

except Exception as e:

print(f"\n❌ Simulation error: {e}")

# For debugging:

import traceback

traceback.print\_exc()

1. row\_navigation.py  
     
   import matplotlib.pyplot as plt

import numpy as np

from scipy.interpolate import CubicSpline

import math

import time

from matplotlib.patches import Polygon

import matplotlib.transforms as transforms

from farm\_safety import SafetyModule

import csv

import os

from datetime import datetime

import random

from collections import OrderedDict

safety = SafetyModule()

STEP = 1.6 # Reduced back to original value for smoother movement (was 0.8)

TOLERANCE = 0.5 # Slightly reduced for better precision

MAX\_ATTEMPTS = 200 # Kept the same

DEBUG = False

ANIMATION\_SPEED = 0.001 # Increased for slowerment

from coordinate\_converter import CoordinateConverter # Import your module

class Rover:

def \_\_init\_\_(self):

self.x = 0.0 # Simulation x (normalized)

self.y = 0.0 # Simulation y (normalized)

self.utm\_x = 0.0 # Real UTM x

self.utm\_y = 0.0 # Real UTM y

self.heading = 0.0

self.last\_heading = 0.0

self.history = []

self.geofence = None

self.entry\_point = None

self.inside\_fence = False

self.command\_count = 0

self.blocked\_directions = set()

self.navigator = None

self.utm\_offset\_x = 0.0 # Offset for normalization

self.utm\_offset\_y = 0.0 # Offset for normalization

self.run\_id = None

self.coord\_converter = CoordinateConverter() # Initialize CoordinateConverter

# Add to Rover class

def log\_movement(self, movement\_type, distance=None, angle=None):

"""Log movement commands to the terminal"""

if movement\_type == "forward":

print(f"⬆️ COMMAND: Move forward {distance:.2f}m")

elif movement\_type == "backward":

print(f"⬇️ COMMAND: Move backward {distance:.2f}m")

elif movement\_type == "turn\_left":

print(f"↩️ COMMAND: Turn left {angle:.1f}°")

elif movement\_type == "turn\_right":

print(f"↪️ COMMAND: Turn right {angle:.1f}°")

elif movement\_type == "stop":

print("🛑 COMMAND: Stop")

self.command\_count += 1

def calculate\_heading\_to(self, tx, ty):

dx, dy = tx - self.x, ty - self.y

if abs(dx)<1e-6 and abs(dy)<1e-6:

return self.heading

ang = math.degrees(math.atan2(dy, dx))

return ang if ang>=0 else ang+360

def set\_position(self, x, y, heading=None, force=False, add\_to\_history=True):

if self.geofence and not force:

in\_fence = self.is\_inside\_farm(x, y)

if not in\_fence:

return False

if self.geofence and not force:

if self.entry\_point and self.distance\_to(\*self.entry\_point) <= 0.8:

self.inside\_fence = True

in\_fence = self.is\_inside\_farm(x, y)

if (in\_fence and not self.inside\_fence) or (not in\_fence and self.inside\_fence):

print("⚠️ Movement blocked: would cross fence boundary")

return False

# Store previous position

prev\_x, prev\_y = self.x, self.y

prev\_heading = self.heading

# Update simulation coordinates (internal use, not logged)

self.x = round(x, 2)

self.y = round(y, 2)

# Calculate real-world UTM coordinates

if hasattr(self, 'navigator') and hasattr(self.navigator, 'utm\_offset\_x'):

self.utm\_x = self.x + self.navigator.utm\_offset\_x

self.utm\_y = self.y + self.navigator.utm\_offset\_y

else:

self.utm\_x = self.x

self.utm\_y = self.y

# Convert UTM to lat/lon

lat, lon = self.coord\_converter.utm\_to\_latlon\_coord(

self.utm\_x, self.utm\_y, zone\_number=45, zone\_letter='N'

)

if lat is None or lon is None:

print("⚠️ Failed to convert UTM to lat/lon, using fallback values")

lat, lon = 0.0, 0.0

if hasattr(self, 'failsafe'):

self.failsafe.update\_gps\_status(has\_fix=True, satellites=10, hdop=1.0)

self.failsafe.update\_internet\_status(connected=True, latency=0.1)

self.failsafe.update\_module\_communication()

if heading is not None:

self.heading = round(heading % 360, 1)

self.last\_heading = self.heading

if add\_to\_history:

self.history.append((self.x, self.y))

self.command\_count += 1

# Get compass direction

compass\_direction = self.get\_compass\_direction(self.heading)

standard\_bearing = (90 - self.heading) % 360

# Terminal output (real-world coordinates only)

print(f"📍 UTM: [{self.utm\_x:.2f}, {self.utm\_y:.2f}] (Zone 45N)")

print(f"📍 Lat/Lon: [{lat:.6f}°N, {lon:.6f}°E]")

print(f"📍 Heading: {self.heading:.1f}° (Compass Heading: {standard\_bearing:.1f}° {compass\_direction})")

# Initialize run\_id if not set or invalid (ensure it's set for first row)

if not hasattr(self, 'run\_id') or self.run\_id is None or self.run\_id == '':

self.run\_id = self.get\_next\_run\_id()

print(f"Debug: Assigned run\_id: {self.run\_id}")

# Prepare log data (real-world coordinates only)

# And add a comment to clarify the units:

log\_data = {

'timestamp': datetime.now().isoformat(),

'run\_id': str(self.run\_id), # Always convert to string, never empty

'x\_utm': self.utm\_x, # Real UTM x

'y\_utm': self.utm\_y, # Real UTM y

'latitude': lat,

'longitude': lon,

'heading': self.heading,

'bearing': standard\_bearing,

'compass\_heading': compass\_direction,

'fix\_quality': '3D Fix',

'satellite\_count': random.randint(8, 12),

'deviation': (self.navigator.calculate\_deviation(self.x, self.y)

if hasattr(self, 'navigator') and self.navigator else 0), # Deviation in centimeters

'data\_age': 0,

'status': 'OK'

}

# Debug: Verify run\_id before logging

deviation\_cm = log\_data['deviation']

if deviation\_cm > 0:

if deviation\_cm < 5:

print(f"📏 Path deviation: {deviation\_cm:.2f} cm (within RTK fixed tolerance)")

else:

print(f"⚠️ Path deviation: {deviation\_cm:.2f} cm (exceeds 5 cm RTK fixed tolerance)")

# Log to CSV

log\_file\_path = r'F:\GPS\task\_2\_waypoints\rover\_log.csv'

with open(log\_file\_path, 'a', newline='') as csvfile:

fieldnames = ['timestamp', 'run\_id', 'x\_utm', 'y\_utm', 'latitude', 'longitude',

'heading', 'bearing', 'compass\_heading', 'fix\_quality', 'satellite\_count',

'deviation', 'data\_age', 'status']

writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

if not os.path.exists(log\_file\_path) or os.path.getsize(log\_file\_path) == 0:

writer.writeheader()

writer.writerow(log\_data)

return True

def get\_compass\_direction(self, heading):

"""

Convert rover's heading (0° = East, 90° = North) to standard compass direction.

Standard compass: North = 0°, East = 90°, South = 180°, West = 270°

"""

# Convert from rover heading to standard compass bearing

compass\_heading = (90 - heading) % 360

# Define standard compass bearings based on converted heading

if 348.75 <= compass\_heading or compass\_heading < 11.25:

return "North"

elif 11.25 <= compass\_heading < 33.75:

return "North-Northeast"

elif 33.75 <= compass\_heading < 56.25:

return "Northeast"

elif 56.25 <= compass\_heading < 78.75:

return "East-Northeast"

elif 78.75 <= compass\_heading < 101.25:

return "East"

elif 101.25 <= compass\_heading < 123.75:

return "East-Southeast"

elif 123.75 <= compass\_heading < 146.25:

return "Southeast"

elif 146.25 <= compass\_heading < 168.75:

return "South-Southeast"

elif 168.75 <= compass\_heading < 191.25:

return "South"

elif 191.25 <= compass\_heading < 213.75:

return "South-Southwest"

elif 213.75 <= compass\_heading < 236.25:

return "Southwest"

elif 236.25 <= compass\_heading < 258.75:

return "West-Southwest"

elif 258.75 <= compass\_heading < 281.25:

return "West"

elif 281.25 <= compass\_heading < 303.75:

return "West-Northwest"

elif 303.75 <= compass\_heading < 326.25:

return "Northwest"

elif 326.25 <= compass\_heading < 348.75:

return "North-Northwest"

else:

return "Unknown" # Should never reach here

def get\_next\_run\_id(self):

"""Determine the next run ID based on existing data in the log file."""

log\_file\_path = r'F:\GPS\task\_2\_waypoints\rover\_log.csv'

# If file doesn't exist, start with run 1

if not os.path.exists(log\_file\_path):

print("Debug: CSV file does not exist, returning run\_id 1")

return 1

try:

# Read the existing file to find the highest run\_id

max\_run\_id = 0

with open(log\_file\_path, 'r', newline='') as csvfile:

reader = csv.DictReader(csvfile)

for row in reader:

if 'run\_id' in row and row['run\_id']:

try:

run\_id = int(row['run\_id'])

max\_run\_id = max(max\_run\_id, run\_id)

except (ValueError, TypeError):

continue

print(f"Debug: Max run\_id found: {max\_run\_id}, returning {max\_run\_id + 1}")

return max\_run\_id + 1

except Exception as e:

print(f"Debug: Error reading run\_id from CSV: {e}, returning 1")

return 1

def set\_geofence(self, vertices, entry\_point):

self.geofence = vertices

self.entry\_point = entry\_point

self.inside\_fence = self.is\_inside\_farm(self.x, self.y)

def distance\_to(self, tx, ty):

return math.hypot(tx - self.x, ty - self.y)

def is\_inside\_farm(self, x, y):

if not self.geofence:

return True

# Simple boundary check

min\_x = min(v[0] for v in self.geofence)

max\_x = max(v[0] for v in self.geofence)

min\_y = min(v[1] for v in self.geofence)

max\_y = max(v[1] for v in self.geofence)

return min\_x <= x <= max\_x and min\_y <= y <= max\_y

def move\_forward(self, distance, ax=None, fig=None, rover\_patch=None):

# Log the movement command to terminal

self.log\_movement("forward", distance=distance)

if self.failsafe.in\_failsafe\_mode:

print("⚠️ Rover is in failsafe mode, cannot move forward.")

return False

rad = math.radians(self.heading)

target\_x = self.x + distance \* math.cos(rad)

target\_y = self.y + distance \* math.sin(rad)

success = self.set\_position(target\_x, target\_y)

if success and ax and fig and rover\_patch:

update\_rover\_visualization(self, ax, fig, rover\_patch)

plt.pause(0.001) # Add pause for smoother animation

return success

# Added method to detect if rover is stuck

def detect\_and\_resolve\_stuck(self):

# If this method is called, we assume potential stuck situation

# Return recommendation for new heading if needed

if len(self.blocked\_directions) > 3:

# Clear blocked directions and suggest a completely new heading

self.blocked\_directions.clear()

# Try perpendicular to current heading

new\_heading = (self.heading + 90) % 360

return new\_heading

return None

# Moved from farm\_entry.py

def update\_rover\_visualization(rover, ax, fig, rover\_patch=None):

rover\_scale = 3.5 # Scale factor to make rover appear larger

if rover\_patch is None:

base\_verts = np.array([[0.7, 0], [-0.3, 0.4], [-0.3, -0.4]])

scaled\_verts = base\_verts \* rover\_scale # Apply scaling

rover\_patch = Polygon(scaled\_verts, closed=True, fc='blue', ec='black') # Correct: uses scaled vertices

ax.add\_patch(rover\_patch)

tr = transforms.Affine2D().rotate\_deg(rover.heading).translate(rover.x, rover.y)

rover\_patch.set\_transform(tr + ax.transData)

if hasattr(ax, 'path\_line') and len(rover.history) > 1:

ax.path\_line.set\_data(\*zip(\*rover.history))

fig.canvas.draw\_idle()

plt.pause(0.001)

return rover\_patch

# Moved from farm\_entry.py

def visualize\_turn(rover, target\_heading, ax, fig, rover\_patch=None, rotation\_speed\_factor=4):

"""

Turn the rover to face a new heading using the shortest possible rotation.

Includes logging for turn diagnostics.

"""

if rover.failsafe.in\_failsafe\_mode:

print("⚠️ Rover is in failsafe mode, cannot turn.")

return rover\_patch

# Normalize headings to [0, 360)

current = rover.heading % 360

target\_heading = target\_heading % 360

# Calculate the absolute angle difference (shortest path)

clockwise\_diff = (target\_heading - current) % 360

counterclockwise\_diff = (current - target\_heading) % 360

# Choose the smallest rotation direction

# FIXED: Swapped direction labels to match standard compass directions

if clockwise\_diff <= counterclockwise\_diff:

angle\_diff = clockwise\_diff

direction = "turn\_left" # Changed from "turn\_right" because clockwise is left in standard compass

else:

angle\_diff = -counterclockwise\_diff

direction = "turn\_right" # Changed from "turn\_left" because counterclockwise is right in standard compass

# Log turn details for debugging

print(f"🔄 TURN: From {current:.1f}° to {target\_heading:.1f}°, Direction: {direction}, Angle: {abs(angle\_diff):.1f}°")

# Skip small turns (threshold 5 degrees)

if abs(angle\_diff) < 5:

rover.heading = target\_heading

rover.last\_heading = target\_heading

print(f"✓ Skipped small turn (<5°), set heading to {target\_heading:.1f}°")

return update\_rover\_visualization(rover, ax, fig, rover\_patch)

# Log the turning command

rover.log\_movement(direction, angle=abs(angle\_diff))

# Calculate steps for smooth animation

steps = max(3, min(int(abs(angle\_diff) / 10), 18))

step\_ang = angle\_diff / steps

# Calculate pause duration

pause\_duration = min(0.01, 0.005 \* (180 / max(1, abs(angle\_diff)))) / rotation\_speed\_factor

# Animate the turn

for i in range(1, steps + 1):

rover.heading = (current + step\_ang \* i) % 360

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

plt.pause(pause\_duration)

# Ensure exact final heading

rover.heading = target\_heading % 360

rover.last\_heading = rover.heading

return update\_rover\_visualization(rover, ax, fig, rover\_patch)

class RowNavigator:

def \_\_init\_\_(self, rover):

self.rover = rover

self.interpolated\_path = []

self.current\_waypoint\_index = 0

self.waypoint\_threshold = 0.02

self.column\_spacing = 1.5

self.column\_height = 15

self.movement\_speed = 1.5

self.current\_row = 0 # Track which row we're on

self.zigzag\_pattern = True # Enable zigzag pattern by default

self.rows\_data = [] # Store information about each row

self.utm\_offset\_x = 0.0 # Initialize UTM offsets

self.utm\_offset\_y = 0.0

def load\_waypoints\_from\_csv(filename):

"""

Load waypoints from a CSV file

Returns a list of (x,y) tuples representing the waypoints path

"""

if not os.path.exists(filename):

print(f"⚠️ Waypoints file not found: {filename}")

return []

waypoints = []

try:

with open(filename, 'r') as csvfile:

reader = csv.DictReader(csvfile)

for row in reader:

x = float(row['x'])

y = float(row['y'])

waypoints.append((x, y))

print(f"✅ Loaded {len(waypoints)} waypoints from {filename}")

return waypoints

except Exception as e:

print(f"❌ Error loading waypoints: {e}")

return []

def load\_rows\_from\_csv(self, csv\_filename):

if not os.path.exists(csv\_filename):

print(f"⚠️ Waypoints file not found: {csv\_filename}")

return False

raw\_points = []

xs, ys = [], []

with open(csv\_filename, 'r') as csvfile:

reader = csv.DictReader(csvfile)

for row in reader:

idx = int(row['row\_index'])

x = float(row['x'])

y = float(row['y'])

raw\_points.append((idx, x, y))

xs.append(x)

ys.append(y)

if not raw\_points:

print("⚠️ No data in CSV.")

return False

# Store UTM offsets for use in coordinate conversion

self.utm\_offset\_x = min(xs)

self.utm\_offset\_y = min(ys)

# Build interpolated\_path with normalized coordinates

self.interpolated\_path = [

(x - self.utm\_offset\_x, y - self.utm\_offset\_y)

for (\_, x, y) in raw\_points

]

# Group by row\_index

grouped = OrderedDict()

for idx, x, y in raw\_points:

grouped.setdefault(idx, []).append((x - self.utm\_offset\_x, y - self.utm\_offset\_y))

# Build rows\_data

self.rows\_data = []

for idx, pts in grouped.items():

start\_pt = pts[0]

end\_pt = pts[-1]

direction = "↑ UP" if end\_pt[1] > start\_pt[1] else "↓ DOWN"

self.rows\_data.append({

'index': idx,

'x\_pos': start\_pt[0],

'direction': direction,

'start': start\_pt,

'end': end\_pt

})

print("\n📋 CSV-based Navigation Plan:")

print(f" Total waypoints: {len(self.interpolated\_path)}")

print(f" Estimated rows: {len(self.rows\_data)}")

print(f" UTM Offsets: ({self.utm\_offset\_x:.2f}, {self.utm\_offset\_y:.2f})")

for row in self.rows\_data:

print(f" Row {row['index']+1}: "

f"X-position {row['x\_pos']:.2f}m, "

f"Direction {row['direction']}")

return True

# Add this method to your RowNavigator class

def calculate\_deviation(self, x, y):

if not self.interpolated\_path or len(self.interpolated\_path) < 2:

return 0

if self.current\_waypoint\_index >= len(self.interpolated\_path) - 1:

# Already at the last waypoint, no current segment

return 0

current\_index = self.current\_waypoint\_index

next\_index = current\_index + 1

p1 = self.interpolated\_path[current\_index]

p2 = self.interpolated\_path[next\_index]

x1, y1 = p1

x2, y2 = p2

# Calculate t (parameter for projection onto the line)

dx = x2 - x1

dy = y2 - y1

if dx == 0 and dy == 0:

# p1 and p2 are the same point, so distance is to that point

return math.hypot(x - x1, y - y1) \* 100 # Convert to cm

l2 = dx\*\*2 + dy\*\*2

t = ((x - x1) \* dx + (y - y1) \* dy) / l2

t = max(0, min(1, t)) # Clamp t to [0, 1]

# Closest point on the segment

closest\_x = x1 + t \* dx

closest\_y = y1 + t \* dy

# Distance from (x, y) to (closest\_x, closest\_y)

distance\_m = math.hypot(x - closest\_x, y - closest\_y)

distance\_cm = distance\_m \* 100 # Convert to cm

return round(distance\_cm, 2)

def generate\_rows(self, start\_x, start\_y, num\_strips=5, strip\_length=None, spacing=None):

if spacing is None:

spacing = self.column\_spacing

if strip\_length is None:

strip\_length = self.column\_height

bottom\_y = start\_y

top\_y = start\_y + strip\_length

self.interpolated\_path = []

self.rows\_data = [] # Reset rows data

print(f"\n🌾 Generating {num\_strips} rows with spacing {spacing:.2f}m")

print(f"🌾 Row length: {strip\_length:.2f}m")

for i in range(num\_strips):

x = start\_x + i \* spacing

# Determine row direction based on zigzag pattern

going\_up = (i % 2 == 0)

direction\_str = "↑ UP" if going\_up else "↓ DOWN"

# Create row points with more intermediate points for smoother movement

if going\_up:

# More points for smoother movement (was 5, now more)

y\_points = np.linspace(bottom\_y, top\_y, 8)

row\_start = (x, bottom\_y)

row\_end = (x, top\_y)

else:

y\_points = np.linspace(top\_y, bottom\_y, 8)

row\_start = (x, top\_y)

row\_end = (x, bottom\_y)

# Store row data for reporting

self.rows\_data.append({

'index': i,

'x\_pos': x,

'direction': direction\_str,

'start': row\_start,

'end': row\_end

})

# Add waypoints for this row

for y in y\_points:

self.interpolated\_path.append((x, y))

# Add improved transition to next row if not the last row

if i < num\_strips - 1:

next\_x = start\_x + (i + 1) \* spacing

transition\_y = top\_y if going\_up else bottom\_y

# Add curved transition points between rows instead of just one point

# This creates a smoother path for the rover to follow

num\_transition\_points = 5 # More points for smoother curve

for j in range(1, num\_transition\_points+1):

t = j / num\_transition\_points

# Create a slight curve for the transition

trans\_x = x + t \* (next\_x - x)

trans\_y = transition\_y

self.interpolated\_path.append((trans\_x, trans\_y))

# Print

# row information

print("\n📋 Row Navigation Plan:")

for row in self.rows\_data:

print(f" Row {row['index']+1}: X-position {row['x\_pos']:.2f}m, Direction {row['direction']}")

return self.interpolated\_path

def distance(self, p1, p2):

return math.hypot(p2[0] - p1[0], p2[1] - p1[1])

def calculate\_heading(self, p1, p2):

"""

Calculate heading between two points with tolerance for small differences.

Modified to handle position rounding precision.

"""

dx = p2[0] - p1[0]

dy = p2[1] - p1[1]

tolerance = 0.01 # Matches position rounding to 2 decimal places

if abs(dx) < tolerance:

print(f"📍 Vertical movement detected: dy={dy:.4f}, setting heading to {'90°' if dy > 0 else '270°'}")

return 90.0 if dy > 0 else 270.0

elif abs(dy) < tolerance:

print(f"📍 Horizontal movement detected: dx={dx:.4f}, setting heading to {'0°' if dx > 0 else '180°'}")

return 0.0 if dx > 0 else 180.0

else:

heading = math.degrees(math.atan2(dy, dx)) % 360

print(f"📍 Diagonal movement: dx={dx:.4f}, dy={dy:.4f}, heading={heading:.1f}°")

return heading

def heading\_difference(self, current, target):

diff = (target - current + 540) % 360 - 180

return diff

def smooth\_turn(self, target\_heading, ax=None, fig=None, rover\_patch=None):

"""

Perform a smooth turn to the target heading.

"""

heading\_diff = self.heading\_difference(self.rover.heading, target\_heading)

if abs(heading\_diff) < 10:

self.rover.heading = target\_heading

self.rover.last\_heading = target\_heading

if ax and fig and rover\_patch:

rover\_patch = update\_rover\_visualization(self.rover, ax, fig, rover\_patch)

print(f"✓ Skipped small turn (<10°), set heading to {target\_heading:.1f}°")

return rover\_patch

direction = "turn\_right" if heading\_diff > 0 else "turn\_left"

self.rover.log\_movement(direction, angle=abs(heading\_diff))

steps = max(5, int(abs(heading\_diff) / 30))

angle\_step = heading\_diff / steps

for i in range(steps):

self.rover.heading = (self.rover.heading + angle\_step) % 360

if ax and fig and rover\_patch:

rover\_patch = update\_rover\_visualization(self.rover, ax, fig, rover\_patch)

plt.pause(0.001)

self.rover.heading = round(target\_heading, 1)

self.rover.last\_heading = self.rover.heading

if ax and fig and rover\_patch:

rover\_patch = update\_rover\_visualization(self.rover, ax, fig, rover\_patch)

return rover\_patch

def move\_precisely\_to\_point(self, target\_point, ax=None, fig=None, rover\_patch=None):

"""

Move rover to target point with precision at constant speed of 1.5 m/s.

Includes logging for turn diagnostics and minimum movement guarantee.

"""

max\_attempts = 40

attempts = 0

last\_time = time.time() # Initialize time tracking

min\_move\_distance = 0.05 # Minimum movement distance to prevent stops

print(f"🎯 Moving to point: ({target\_point[0]:.2f}, {target\_point[1]:.2f}) at {self.movement\_speed} m/s")

while attempts < max\_attempts:

if self.rover.failsafe.in\_failsafe\_mode:

print("⚠️ Rover is in failsafe mode, stopping movement.")

return False

# Calculate time elapsed with minimum threshold

current\_time = time.time()

time\_elapsed = current\_time - last\_time

last\_time = current\_time

# Ensure minimum time step to prevent tiny movements

if time\_elapsed < 0.01:

time\_elapsed = 0.05 # Minimum time step

current\_pos = (self.rover.x, self.rover.y)

dist\_to\_target = self.distance(current\_pos, target\_point)

if attempts % 5 == 0:

print(f" Distance to target: {dist\_to\_target:.2f}m")

if dist\_to\_target <= self.waypoint\_threshold:

print(f"✅ Reached target within {self.waypoint\_threshold}m threshold")

self.rover.set\_position(target\_point[0], target\_point[1], force=True)

if ax and fig and rover\_patch:

rover\_patch = update\_rover\_visualization(self.rover, ax, fig, rover\_patch)

return True

desired\_heading = self.calculate\_heading(current\_pos, target\_point)

# Only turn if heading difference is significant

heading\_diff = self.heading\_difference(self.rover.heading, desired\_heading)

if abs(heading\_diff) > 10:

print(f"🧭 Initiating turn: Current {self.rover.heading:.1f}°, Desired {desired\_heading:.1f}°, Diff {heading\_diff:.1f}°")

self.smooth\_turn(desired\_heading, ax, fig, rover\_patch)

else:

print(f"✓ Heading diff {abs(heading\_diff):.1f}° < 10°, no turn needed")

# Calculate move\_dist based on constant speed (1.5 m/s) and elapsed time

move\_dist = self.movement\_speed \* time\_elapsed

# Ensure minimum movement distance

if move\_dist < min\_move\_distance:

move\_dist = min\_move\_distance

# Don't overshoot target

if move\_dist > dist\_to\_target:

move\_dist = dist\_to\_target

path = [(self.rover.x, self.rover.y),

(self.rover.x + move\_dist \* math.cos(math.radians(self.rover.heading)),

self.rover.y + move\_dist \* math.sin(math.radians(self.rover.heading)))]

status, \_ = safety.check\_safety([self.rover.x, self.rover.y], self.rover.heading, path)

if status == 'safe':

success = self.rover.move\_forward(move\_dist, ax, fig, rover\_patch)

if not success:

print("⚠️ Movement failed - obstacle detected")

return False

else:

print(f"⚠️ Safety check failed: {status}")

return False

attempts += 1

# Ensure consistent visualization frequency

if ax and fig:

plt.pause(0.001)

print("⚠️ Max attempts reached")

return False

def navigate\_to\_starting\_point(self, ax=None, fig=None, rover\_patch=None):

if not self.interpolated\_path:

return False

starting\_point = self.interpolated\_path[0]

# Enhanced terminal output (point 4.i)

print(f"\n🚩 Starting point: ({starting\_point[0]:.2f}, {starting\_point[1]:.2f})")

if self.rows\_data:

print(f"🌾 This is the beginning of Row 1, moving {self.rows\_data[0]['direction']}")

# Navigate to starting point

result = self.move\_precisely\_to\_point(starting\_point, ax, fig, rover\_patch)

if result:

self.current\_waypoint\_index = 0

self.current\_row = 0

print(f"✅ Successfully reached Row 1 starting point")

return result

def determine\_next\_task(self):

if not self.interpolated\_path or self.current\_waypoint\_index >= len(self.interpolated\_path) - 1:

return None

next\_idx = self.current\_waypoint\_index + 1

return self.interpolated\_path[next\_idx]

def align\_to\_next\_task(self, ax=None, fig=None, rover\_patch=None):

"""

Align rover to the next waypoint.

"""

next\_point = self.determine\_next\_task()

if not next\_point:

return False

current\_pos = (self.rover.x, self.rover.y)

desired\_heading = self.calculate\_heading(current\_pos, next\_point)

current\_heading = self.rover.heading

heading\_diff = self.heading\_difference(current\_heading, desired\_heading)

if abs(heading\_diff) < 10:

print(f"✓ Already aligned to correct heading: {current\_heading:.1f}°")

return True

print(f"🧭 Aligning from {current\_heading:.1f}° to {desired\_heading:.1f}°")

if ax and fig:

rover\_patch = visualize\_turn(self.rover, desired\_heading, ax, fig, rover\_patch, rotation\_speed\_factor=6)

else:

self.rover.heading = desired\_heading

self.rover.last\_heading = desired\_heading

print(f"✓ Aligned to heading: {self.rover.heading:.1f}°")

return True

def navigate\_all\_rows(self, ax=None, fig=None, rover\_patch=None):

if not self.interpolated\_path:

print("⚠️ No path generated - call generate\_rows first")

return False

if not self.navigate\_to\_starting\_point(ax, fig, rover\_patch):

return False

while self.current\_waypoint\_index < len(self.interpolated\_path) - 1:

next\_point = self.determine\_next\_task()

if not next\_point:

break

if not self.align\_to\_next\_task(ax, fig, rover\_patch):

continue

if self.move\_precisely\_to\_point(next\_point, ax, fig, rover\_patch):

self.current\_waypoint\_index += 1

return True

def navigate\_path(self, ax=None, fig=None, rover\_patch=None):

"""

Navigate through all waypoints with improved turning and reporting.

"""

# 1) Precompute total points

total\_pts = len(self.interpolated\_path)

print(f"🚜 COMMAND: Navigate zigzag path with {total\_pts} points")

if total\_pts == 0:

print("⚠️ No path generated - call generate\_rows or load\_rows first")

return False

if self.current\_waypoint\_index >= total\_pts:

print("⚠️ Navigation complete - already at end of path")

return True

# 2) Compute a row‐matching tolerance halfway between your actual row y‐positions

# Rows are differentiated by their y-coordinates in zigzag pattern

ys = sorted({pt[1] for pt in self.interpolated\_path})

unique\_ys = []

# Filter out very close y values (within 0.1)

for y in ys:

if not unique\_ys or abs(y - unique\_ys[-1]) > 0.1:

unique\_ys.append(y)

if len(unique\_ys) > 1:

tol = min(abs(unique\_ys[i+1] - unique\_ys[i]) for i in range(len(unique\_ys)-1)) / 2

else:

tol = self.column\_spacing / 2

# 3) Helper to find which row a point belongs to - using y-coordinate and REVERSE indexing

def find\_row(point):

# Match point to row based on y-coordinate with REVERSED indexing

for idx, row\_y in enumerate(unique\_ys):

if abs(point[1] - row\_y) <= tol:

# Return reversed index (if we have n rows, index should be n-idx-1)

return len(unique\_ys) - idx - 1

return len(unique\_ys) - 1 # Default to last row if no match

# 4) Log planned moves once, not in the loop

current\_pos = self.interpolated\_path[self.current\_waypoint\_index]

current\_row\_idx = find\_row(current\_pos)

print(f"\n🌾 Starting in Row {current\_row\_idx+1}")

# Set initial row direction

if current\_row\_idx < len(self.rows\_data):

current\_direction = self.rows\_data[current\_row\_idx]['direction']

else:

# Infer direction from path if row data is incomplete

next\_pt = self.interpolated\_path[self.current\_waypoint\_index + 1] if self.current\_waypoint\_index + 1 < total\_pts else None

if next\_pt:

if next\_pt[0] > current\_pos[0]:

current\_direction = "→ East"

elif next\_pt[0] < current\_pos[0]:

current\_direction = "← West"

else:

current\_direction = "Unknown"

else:

current\_direction = "Unknown"

print(f"🧭 Current direction: {current\_direction}")

success = True

last\_command\_point = None

last\_target\_point = None # Track the last target point to avoid duplicate commands

# 6) Main loop

while self.current\_waypoint\_index < total\_pts - 1:

if self.rover.failsafe.in\_failsafe\_mode:

print("⚠️ Rover is in failsafe mode, stopping navigation.")

return False

next\_idx = self.current\_waypoint\_index + 1

next\_pt = self.interpolated\_path[next\_idx]

current\_pt = self.interpolated\_path[self.current\_waypoint\_index]

# Only print movement command if it's a new target point (avoid duplicates)

if next\_pt != last\_target\_point:

print(f"\n➡️ COMMAND: Move from ({current\_pt[0]:.2f}, {current\_pt[1]:.2f}) "

f"to ({next\_pt[0]:.2f}, {next\_pt[1]:.2f})")

last\_target\_point = next\_pt

last\_command\_point = current\_pt

# Detect if row changes by comparing y-coordinates

new\_row\_idx = find\_row(next\_pt)

if new\_row\_idx != current\_row\_idx:

print(f"\n🔄 Transitioning from Row {current\_row\_idx+1} to Row {new\_row\_idx+1}")

# Get or infer new direction

if new\_row\_idx < len(self.rows\_data):

new\_direction = self.rows\_data[new\_row\_idx]['direction']

else:

# Infer direction from next points in path

future\_idx = next\_idx + 1

if future\_idx < total\_pts:

future\_pt = self.interpolated\_path[future\_idx]

if future\_pt[0] > next\_pt[0]:

new\_direction = "→ East"

elif future\_pt[0] < next\_pt[0]:

new\_direction = "← West"

else:

new\_direction = "Unknown"

else:

new\_direction = "Unknown"

print(f"🧭 New row direction: {new\_direction}")

# Plan turn for the entire row

if new\_row\_idx < len(self.rows\_data):

start, end = self.rows\_data[new\_row\_idx]['start'], self.rows\_data[new\_row\_idx]['end']

if self.rows\_data[new\_row\_idx]['direction'].startswith("↑") or self.rows\_data[new\_row\_idx]['direction'].startswith("→"):

optimal = self.calculate\_heading(start, end)

else:

optimal = self.calculate\_heading(end, start)

print(f"🧭 Planning efficient turn from {self.rover.heading:.1f}° to {optimal:.1f}°")

if not self.align\_to\_next\_task(ax, fig, rover\_patch):

print("⚠️ Failed to align for transition")

success = False

break

current\_row\_idx = new\_row\_idx

current\_direction = new\_direction

else:

# Only print "continuing in row" message when we're staying in the same row

# and moving to a new target point (not repeatedly)

if next\_pt != last\_target\_point:

print(f"\n🌾 Continuing in Row {current\_row\_idx+1} moving {current\_direction}")

# Check if we need to adjust heading within the row

desired = self.calculate\_heading((self.rover.x, self.rover.y), next\_pt)

if abs(self.heading\_difference(self.rover.heading, desired)) > 3:

if not self.align\_to\_next\_task(ax, fig, rover\_patch):

print("⚠️ Failed to align in‐row")

success = False

break

# Drive to the point

if self.move\_precisely\_to\_point(next\_pt, ax, fig, rover\_patch):

self.current\_waypoint\_index = next\_idx

print(f"✅ Reached waypoint {self.current\_waypoint\_index+1} in Row {current\_row\_idx+1}")

else:

print(f"⚠️ Failed to reach waypoint {next\_idx+1}")

success = False

break

if ax and fig:

plt.pause(0.05)

# 7) Final summary if we made it

if success and self.current\_waypoint\_index == total\_pts - 1:

print("\n✅ Successfully navigated entire path")

final\_row = find\_row(self.interpolated\_path[-1])

print(f"🎉 Completed Row {final\_row+1} – All rows navigated!")

# ensure exact final positioning

fx, fy = self.interpolated\_path[-1]

if self.rover.distance\_to(fx, fy) > self.waypoint\_threshold:

print(f"📍 Final adjustment to ({fx:.2f}, {fy:.2f})...")

self.rover.set\_position(fx, fy, force=True)

if ax and fig and rover\_patch:

rover\_patch = update\_rover\_visualization(self.rover, ax, fig, rover\_patch)

plt.pause(0.5)

return success

def find\_best\_path\_angle(rover, tx, ty, blocked\_angles=None):

direct = math.degrees(math.atan2(ty - rover.y, tx - rover.x)) % 360

if not blocked\_angles or int(direct/10)\*10 not in blocked\_angles:

return direct

for off in range(10,360,10):

for sign in (1,-1):

ta = (direct+sign\*off)%360

if int(ta/10)\*10 not in blocked\_angles:

return ta

import random; return random.randint(0,359)

def normalize\_coordinates(waypoints):

"""

Transform coordinates from large values (like UTM) to simulation coordinate space

"""

if not waypoints:

return []

# Find min values to use as origin

min\_x = min(point[0] for point in waypoints)

min\_y = min(point[1] for point in waypoints)

# Normalize all points relative to this origin

normalized = []

for x, y in waypoints:

normalized.append((x - min\_x, y - min\_y))

print(f"✓ Normalized coordinates from ({min\_x:.1f}, {min\_y:.1f}) origin")

return normalized

def get\_float(prompt):

while True:

try:

return float(input(prompt))

except ValueError:

print("⚠️ Please enter a valid number.")

def navigate\_to\_point(rover, tx, ty, ax, fig, rover\_patch=None, step\_size=STEP, tolerance=TOLERANCE):

"""

Navigate to a specific point using the rover's defined movement speed

"""

# Use rover's movement speed if step\_size not specified

if step\_size is None and hasattr(rover, 'navigator') and hasattr(rover.navigator, 'movement\_speed'):

step\_size = rover.navigator.movement\_speed

else:

step\_size = STEP # Fall back to default if not availabl

print(f"\n🚗 Navigating to point ({tx:.3f}, {ty:.3f})...\n")

dist = rover.distance\_to(tx,ty)

attempts=0; last\_dist=float('inf'); alt=False; blocked=0

while dist>tolerance and attempts<MAX\_ATTEMPTS:

if rover.failsafe.in\_failsafe\_mode:

print("⚠️ Rover is in failsafe mode, stopping navigation.")

return False, rover\_patch

attempts+=1

rec = rover.detect\_and\_resolve\_stuck()

if rec is not None:

rover\_patch = visualize\_turn(rover, rec, ax, fig, rover\_patch)

alt=True; continue

if attempts%5==0:

if dist>last\_dist\*0.95 and not alt:

print("⚠️ Limited progress detected, trying alternative approach...")

rover.blocked\_directions.clear()

angle=(rover.heading+90+attempts%90)%360

rover\_patch=visualize\_turn(rover,angle,ax,fig,rover\_patch)

step\_size=min(step\_size\*2,dist/2); alt=True

else:

alt=False; step\_size=min(STEP,dist/2)

last\_dist=dist

if blocked>2:

tgt=find\_best\_path\_angle(rover,tx,ty,rover.blocked\_directions)

blocked=0

else:

tgt=rover.calculate\_heading\_to(tx,ty)

diff=(tgt-rover.heading+180)%360-180

if abs(diff)>5:

rover\_patch=visualize\_turn(rover,tgt,ax,fig,rover\_patch)

step=min(step\_size,dist)

# Set up movement parameters

target\_x = rover.x + step \* math.cos(math.radians(rover.heading))

target\_y = rover.y + step \* math.sin(math.radians(rover.heading))

path = [(rover.x, rover.y), (target\_x, target\_y)]

# Safety check before movement

status, recovery\_data = safety.check\_safety([rover.x, rover.y], rover.heading, path)

ok = False # Default to unsuccessful movement

if status == 'safe':

# Safe to proceed with normal movement

ok = rover.move\_forward(step, ax, fig, rover\_patch)

elif status == 'drift':

# Handle drift scenario

pos, heading, drift\_status, updated\_data = safety.handle\_drift(

[rover.x, rover.y], rover.heading, recovery\_data)

# Update rover position and visualize

rover.set\_position(pos[0], pos[1], heading, add\_to\_history=True)

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

# Update drift data or clear it if recovered

if drift\_status == 'recovered':

ok = True

else:

recovery\_data = updated\_data

ok = False

blocked += 1

elif status in ['no-go', 'outside']:

# Handle no-go zone or boundary violation

pos, heading, violation\_status = safety.handle\_no\_go\_violation(

[rover.x, rover.y], rover.heading, recovery\_data)

# Update rover position and visualize

rover.set\_position(pos[0], pos[1], heading, add\_to\_history=True)

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

if violation\_status == 'recovered':

ok = True

else:

ok = False

blocked += 1

# Update visualization

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

dist = rover.distance\_to(tx, ty)

if not ok:

blocked += 1

if blocked >= 2:

ch = 45 + blocked \* 15

ch = min(ch, 180)

rover\_patch = visualize\_turn(rover, (rover.heading + ch) % 360, ax, fig, rover\_patch)

else:

blocked = 0

if dist <= tolerance:

print(f"✅ Reached target point ({rover.x:.3f}, {rover.y:.3f})")

return True, rover\_patch

print("🔄 Making final approach attempt with larger step size...")

direct = rover.calculate\_heading\_to(tx, ty)

rover\_patch = visualize\_turn(rover, direct, ax, fig, rover\_patch)

rover.move\_forward(dist \* 0.9, ax, fig, rover\_patch)

fd = rover.distance\_to(tx, ty)

if fd <= tolerance \* 1.5:

print(f"✅ Reached target point on final attempt ({rover.x:.3f}, {rover.y:.3f})")

return True, rover\_patch

print(f"⚠️ Could not reach target point. Current position: ({rover.x:.3f}, {rover.y:.3f})")

print(f" Distance to target: {fd:.3f}")

return False, rover\_patch

def follow\_path\_precisely(rover, waypoints, ax, fig, rover\_patch):

"""

Follows the planned path with ultra-high precision by enforcing strict path adherence

Args:

rover: Rover instance

waypoints: List of (x,y) points to follow

ax: Matplotlib axis

fig: Matplotlib figure

rover\_patch: Visual representation of rover

Returns:

bool: True if path followed successfully, False otherwise

rover\_patch: Updated rover patch

"""

print(f"🛣️ COMMAND: Follow path with {len(waypoints)} waypoints")

if not waypoints or len(waypoints) < 2:

print("⚠️ Path too short or empty")

return False, rover\_patch

print(f"\n🛣️ Following planned path with {len(waypoints)} waypoints...")

# Constants for strict path following - adjusted for speed

PATH\_STEP = rover.navigator.movement\_speed # Increased step size for faster movement (was 0.2)

PATH\_TOLERANCE = 0.05 # Small tolerance to enforce strict adherence

ANIMATION\_SPEED = 0.001 # Faster animation (was 0.01)

ROTATION\_STEP\_FACTOR = 8 # Rotate faster

# Start with current position

start\_idx = 0

# Find closest waypoint if we're not already at the first one

if rover.distance\_to(\*waypoints[0]) > PATH\_TOLERANCE:

closest\_idx = 0

min\_dist = float('inf')

for i, wp in enumerate(waypoints):

if rover.failsafe.in\_failsafe\_mode:

print("⚠️ Rover is in failsafe mode, stopping path following.")

return False, rover\_patch

dist = rover.distance\_to(\*wp)

if dist < min\_dist:

min\_dist = dist

closest\_idx = i

# If we're closer to a waypoint further along the path, start from there

if closest\_idx > 0 and min\_dist < PATH\_TOLERANCE:

start\_idx = closest\_idx

print(f"Starting from waypoint {start\_idx} which is closest to current position")

else:

# We need to first move to the first waypoint

print(f"Moving to the first waypoint at ({waypoints[0][0]:.2f}, {waypoints[0][1]:.2f})")

initial\_heading = rover.calculate\_heading\_to(\*waypoints[0])

rover\_patch = visualize\_turn(rover, initial\_heading, ax, fig, rover\_patch, rotation\_speed\_factor=ROTATION\_STEP\_FACTOR)

# Don't teleport - move properly to first waypoint

init\_distance = rover.distance\_to(\*waypoints[0])

if init\_distance > PATH\_TOLERANCE:

segments = max(2, int(init\_distance / PATH\_STEP))

step\_dist = init\_distance / segments

for \_ in range(segments):

success = rover.move\_forward(step\_dist, ax, fig, rover\_patch)

if not success:

# If blocked, try with smaller steps

half\_step = step\_dist / 2

if half\_step > 0.1: # Don't try with too small steps

success = rover.move\_forward(half\_step, ax, fig, rover\_patch)

rover\_patch = update\_rover\_visualization(rover, ax, fig, rover\_patch)

plt.pause(ANIMATION\_SPEED)

# For path visualization

actual\_path = []

path\_line = None

# Traverse waypoints

for i in range(start\_idx, len(waypoints)-1):

current\_wp = waypoints[i]

next\_wp = waypoints[i+1]

print(f"\n📍 Moving from waypoint {i} to {i+1}: ({current\_wp[0]:.2f}, {current\_wp[1]:.2f}) → ({next\_wp[0]:.2f}, {next\_wp[1]:.2f})")

# Calculate segment vector and length

segment\_vec = (next\_wp[0] - current\_wp[0], next\_wp[1] - current\_wp[1])

segment\_len = math.hypot(\*segment\_vec)

if segment\_len < 0.01: # Skip tiny segments

continue

# Unit vector along segment

unit\_vec = (segment\_vec[0]/segment\_len, segment\_vec[1]/segment\_len)

# Align precisely to segment direction with faster rotation

segment\_heading = math.degrees(math.atan2(segment\_vec[1], segment\_vec[0])) % 360

rover\_patch = visualize\_turn(rover, segment\_heading, ax, fig, rover\_patch, rotation\_speed\_factor=ROTATION\_STEP\_FACTOR)

# Before starting segment, ensure we're exactly at the start point (if not already there)

if rover.distance\_to(\*current\_wp) > PATH\_TOLERANCE:

# Move to start point without teleporting

remaining\_dist = rover.distance\_to(\*current\_wp)

1. farm\_safety.py

import math

import random

class SafetyModule:

"""

A module to handle drift and enforce no-go rules in farm rover navigation.

No-go now covers:

1) Revisiting a waypoint (in the user-defined waypoint list).

2) Exiting the farm boundary.

Drift logic is unchanged.

"""

def \_\_init\_\_(self, failsafe = None, revisit\_threshold=0.2):

# Drift configuration

self.failsafe = failsafe # Link to FailsafeModule

self.is\_paused = False

self.turn\_rate\_per\_cm = 20 # degrees per cm of turn capability

self.distance\_per\_step = 0.2 # cm traveled per simulation step

self.lookahead\_steps = 5 # baseline lookahead for drift recovery

self.drift\_probability = 0.05 # probability of drift occurring per check

# Geofence

self.geofence = None # farm boundary polygon vertices

# Waypoint tracking for revisit prevention

self.waypoints = [] # list of (x,y) waypoints

self.visited\_wp\_indices = set() # indices of waypoints already visited

self.revisit\_threshold = revisit\_threshold

# History logs

self.violations\_history = [] # track no-go incidents

self.drift\_history = [] # track drift incidents

def on\_failsafe\_triggered(self):

self.is\_paused = True # Pause SafetyModule

def on\_failsafe\_cleared(self):

self.is\_paused = False # Resume SafetyModule

def set\_geofence(self, vertices):

"""Set the farm boundary as a polygon"""

self.geofence = vertices

def set\_waypoints(self, waypoints):

"""Provide the ordered list of navigation waypoints"""

self.waypoints = waypoints

self.visited\_wp\_indices.clear()

def is\_outside\_geofence(self, pos):

"""Check if a position is outside the farm boundary polygon"""

if not self.geofence:

return False

x, y = pos

inside = False

n = len(self.geofence)

p1x, p1y = self.geofence[0]

for i in range(1, n+1):

p2x, p2y = self.geofence[i % n]

if y > min(p1y, p2y):

if y <= max(p1y, p2y) and x <= max(p1x, p2x):

if p1y != p2y:

xinters = (y - p1y) \* (p2x - p1x) / (p2y - p1y) + p1x

if p1x == p2x or x <= xinters:

inside = not inside

p1x, p1y = p2x, p2y

return not inside

def calculate\_drift(self, pos):

"""Calculate the drift distance (placeholder implementation)."""

# TODO: Implement actual drift calculation based on requirements

return 0.0 # Placeholder; replace with real logic

def adjust\_path(self, pos, path):

"""Adjust the path to account for drift (placeholder implementation)."""

# TODO: Implement path adjustment logic

return path # Placeholder; replace with real logic

def check\_safety(self, pos, heading, path):

"""

Check if a planned move is safe.

Returns:

- status: 'safe', 'drift', or 'no-go'

- data: info for recovery or violation handling

"""

if self.is\_paused: # If paused by Failsafe

return 'safe', None # Do nothing

if self.failsafe is not None and self.failsafe.in\_failsafe\_mode: # Check if Failsafe is active # Check if Failsafe is active

return 'safe', None # Do nothing if it is

# Candidate next position

next\_pos = path[-1]

# 1) No-go: revisiting an already visited waypoint

for idx, wp in enumerate(self.waypoints):

if math.hypot(next\_pos[0]-wp[0], next\_pos[1]-wp[1]) <= self.revisit\_threshold:

if idx in self.visited\_wp\_indices:

self.violations\_history.append(('no-go-revisit', next\_pos, heading))

return 'no-go', {

'violation\_type': 'revisit',

'pos': next\_pos,

'heading': heading

}

else:

# Mark this waypoint as visited now

self.visited\_wp\_indices.add(idx)

break

# 2) No-go: exiting farm boundary

if self.is\_outside\_geofence(next\_pos):

self.violations\_history.append(('no-go-boundary', next\_pos, heading))

return 'no-go', {

'violation\_type': 'boundary',

'pos': next\_pos,

'heading': heading

}

# 3) Potential drift

drift\_distance = self.calculate\_drift(pos) # Use pos instead of current\_position

if drift\_distance > 1.0: # If drift is too big

self.failsafe.trigger\_custom\_failsafe("Severe drift detected") # Call Failsafe

return 'safe', None

elif drift\_distance > 0.5: # Small drift

new\_path = self.adjust\_path(pos, path) # Compute adjusted path

return 'adjusted', new\_path

# Other safety checks

if path and random.random() < self.drift\_probability:

closest\_idx, closest\_point = self.find\_closest\_point\_on\_path(pos, path)

if closest\_idx < len(path) - self.lookahead\_steps:

# Calculate drift distance

drift\_distance = math.hypot(pos[0] - closest\_point[0], pos[1] - closest\_point[1])

self.drift\_history.append(('drift', pos.copy(), heading))

# Report to FailsafeModule if provided

if self.failsafe:

drift\_distance, severity = self.failsafe.report\_drift(pos, closest\_point)

if severity in ['severe', 'critical']: # Adjust based on actual severity enum/values

return 'drift', {'severity': severity, 'distance': drift\_distance}

# Proceed with SafetyModule recovery for minor/moderate drift

drift\_angle = 45 if random.choice([True, False]) else -45

trigger\_idx = closest\_idx

end\_idx = min(trigger\_idx + self.lookahead\_steps, len(path) - 1)

dx = path[end\_idx][0] - path[trigger\_idx][0]

dy = path[end\_idx][1] - path[trigger\_idx][1]

mag = math.hypot(dx, dy)

ux, uy = (dx / mag, dy / mag) if mag else (1.0, 0.0)

rad = math.radians(drift\_angle)

c, s = math.cos(rad), math.sin(rad)

rx = ux \* c + uy \* s

ry = -ux \* s + uy \* c

turn\_dist = abs(drift\_angle) / self.turn\_rate\_per\_cm

extra\_skip = int(turn\_dist / self.distance\_per\_step)

recovery\_idx = min(trigger\_idx + self.lookahead\_steps + extra\_skip, len(path) - 1)

recovery\_target = path[recovery\_idx]

return 'drift', {

'trigger\_idx': trigger\_idx,

'drift\_angle': drift\_angle,

'drift\_vector': (rx, ry),

'recovery\_idx': recovery\_idx,

'recovery\_target': recovery\_target,

'path': path,

'distance': drift\_distance

}

# 4) Safe to proceed

return 'safe', None

def handle\_drift(self, pos, heading, drift\_data):

"""Simulate drift and guide recovery."""

# TODO: Implement drift handling logic

raise NotImplementedError

def handle\_no\_go\_violation(self, pos, heading, violation\_data):

"""Back away from forbidden position."""

# TODO: Implement no-go violation handling logic

raise NotImplementedError

def find\_closest\_point\_on\_path(self, pos, path):

"""Find the path index closest to pos."""

min\_dist, min\_idx = float('inf'), 0

for i, p in enumerate(path):

d = math.hypot(p[0]-pos[0], p[1]-pos[1])

if d < min\_dist:

min\_dist, min\_idx = d, i

return min\_idx, path[min\_idx]

def diff\_h(self, c, t):

return (t - c + 540) % 360 - 180

4.rover\_health\_check.py

"""

Integrated Rover Health Check Module for Emlid Reach M2 RTK GPS

This module combines the RoverHealthCheck and RTKGPSRover classes

to perform health checks using the existing rover interface.

"""

import time

import logging

import os

import math

import serial

import pynmea2

import pyproj

from pyproj import Proj

import numpy as np

from datetime import datetime, timezone

import csv

# Configure logging

LOG\_PATH = r'F:\GPS\task\_2\_waypoints\rover\_health.csv'

os.makedirs(os.path.dirname(LOG\_PATH), exist\_ok=True)

logger = logging.getLogger('rover\_health')

logger.setLevel(logging.INFO)

if not any(isinstance(h, logging.FileHandler) and h.baseFilename == LOG\_PATH

for h in logger.handlers):

fh = logging.FileHandler(LOG\_PATH, mode='a', encoding='utf-8')

fmt = logging.Formatter('%(asctime)s,%(levelname)s,%(message)s')

fh.setFormatter(fmt)

logger.addHandler(fh)

class HealthCheckFailure(Exception):

"""Exception raised when a health check fails."""

pass

class RTKGPSRover:

"""Base class for RTK GPS functionality to be used with the health check module."""

def \_\_init\_\_(self, port='COM8', baudrate=115200, log\_data=True, log\_path='gps\_logs'):

# Configure the serial connection

try:

self.ser = serial.Serial(

port=port,

baudrate=baudrate,

timeout=1

)

print(f"Connected to: {self.ser.port}")

except serial.SerialException:

print(f"Warning: Could not connect to serial port {port}. Using simulation mode.")

self.ser = None

# Data storage

self.latitude = 0.0

self.longitude = 0.0

self.altitude = 0.0

self.fix\_quality = 4 # Default to RTK Fixed for simulation

self.satellites = 10 # Default satellite count for simulation

self.hdop = 1.0 # Default HDOP for simulation

self.speed = 0.0 # in knots

self.course = 0.0 # in degrees

self.fix\_time = datetime.now().strftime('%H:%M:%S')

self.log\_data = log\_data

# Quality indicator strings

self.fix\_quality\_str = {

0: "Invalid",

1: "GPS Fix",

2: "DGPS Fix",

4: "RTK Fixed",

5: "RTK Float",

6: "Estimated (DR) Fix"

}

# Additional fields for health check

self.pdop = 1.2 # Default value for simulation

self.vdop = 1.0 # Default value for simulation

self.age\_of\_corrections = 1.0 # Default for simulation

self.age\_of\_corrections\_ms = 50 # Default in milliseconds for simulation

self.satellites\_data = [{'prn': f'{i}', 'elevation': 45+i, 'snr': 45+i} for i in range(10)]

self.constellations = ['GPS', 'GLONASS'] # Default for simulation

self.average\_snr = 45 # Default for simulation

self.min\_elevation = 15 # Default for simulation

self.time\_diff = 0.02 # Default for simulation

self.time\_diff\_ns = 20 # Default in nanoseconds for simulation

self.pps\_jitter\_ns = 15 # Default PPS jitter in nanoseconds

self.easting = 500000 # Default for simulation

self.northing = 3000000 # Default for simulation

self.rtk\_init\_time = 15 # Default RTK initialization time in seconds

# Farm boundary coordinates (configurable)

self.LAT\_MIN, self.LAT\_MAX = 12.345, 12.678

self.LON\_MIN, self.LON\_MAX = 76.543, 76.876

# Logging setup

if log\_data and not os.path.exists(log\_path):

os.makedirs(log\_path)

print(f"RTK GPS Rover initialized. {'Logging enabled.' if log\_data else 'Logging disabled.'}")

def read\_nmea\_data(self, num\_lines=10, timeout=1):

"""

Read NMEA data from serial port or simulate it.

For simulation purposes, we'll just return the default values.

"""

if self.ser is None:

# Simulation mode - return default values

return {'GGA': ['simulated'], 'GSA': ['simulated'], 'GSV': ['simulated'], 'RMC': ['simulated']}

# Real mode implementation would go here...

# For simulation, just return simulated data

return {'GGA': ['simulated'], 'GSA': ['simulated'], 'GSV': ['simulated'], 'RMC': ['simulated']}

class RoverHealthCheck:

"""Performs comprehensive health checks on the rover's RTK GPS system."""

def \_\_init\_\_(self, rover):

"""

Initialize health check parameters using an existing rover instance.

Args:

rover (Rover): The rover instance to check

"""

# Create our RTK GPS rover instance for health checks

self.rtk\_rover = RTKGPSRover()

# Store reference to the main rover

self.rover = rover

# Define health status dictionary

self.health\_status = {

'rtk\_status': False,

'satellite\_count': False,

'dop\_values': False,

'signal\_strength': False,

'age\_of\_corrections': False,

'position\_validity': False,

'constellation\_diversity': False,

'elevation\_mask': False,

'rtk\_init\_time': False,

'position\_stability': False,

'antenna\_placement': False,

'coordinate\_system': False,

'multipath\_detection': False,

'gps\_sync\_time': False,

'receiver\_clock\_stability': False,

'power\_supply': False,

'firmware\_updates': False,

'battery\_level': False,

'hardware\_status': False

}

def check\_antenna\_placement(self):

"""Check antenna placement by verifying satellite count and signal strength."""

satellites = self.rtk\_rover.satellites

average\_snr = self.rtk\_rover.average\_snr

if satellites < 6 or average\_snr < 35:

logger.warning(f"Low signal: satellites={satellites}, SNR={average\_snr:.1f} dB-Hz")

time.sleep(1) # Reduced sleep time for simulation

else:

logger.info(f"Antenna placement OK: satellites={satellites}, SNR={average\_snr:.1f} dB-Hz")

self.health\_status['antenna\_placement'] = True

def check\_rtk\_status(self):

"""Check RTK fix status for optimal accuracy."""

fix\_status = self.rtk\_rover.fix\_quality

if fix\_status == 4:

logger.info("RTK status: Fixed")

elif fix\_status == 5:

logger.warning("RTK status: Float")

time.sleep(1) # Reduced sleep time for simulation

elif fix\_status == 1:

logger.error("RTK status: Single")

# For simulation, we'll continue rather than failing

# raise HealthCheckFailure("RTK status is Single")

else:

logger.warning(f"RTK status: {fix\_status}")

time.sleep(1) # Reduced sleep time for simulation

self.health\_status['rtk\_status'] = True

def check\_satellite\_count(self):

"""Verify sufficient satellites for stable RTK fix."""

satellites = self.rtk\_rover.satellites

if satellites < 4:

logger.error(f"Only {satellites} satellites")

# For simulation, we'll continue rather than failing

# raise HealthCheckFailure("Insufficient satellites")

elif satellites < 6:

logger.warning(f"Only {satellites} satellites")

time.sleep(1) # Reduced sleep time for simulation

else:

logger.info(f"Satellite count OK: {satellites} satellites")

self.health\_status['satellite\_count'] = True

def check\_dop\_values(self):

"""Check DOP values for optimal satellite geometry."""

pdop = self.rtk\_rover.pdop

hdop = self.rtk\_rover.hdop

vdop = self.rtk\_rover.vdop

if pdop is None or hdop is None or vdop is None:

logger.warning("DOP values not available")

return

max\_dop = max(pdop, hdop, vdop)

if max\_dop > 2.0:

logger.error(f"High DOP: PDOP={pdop:.1f}, HDOP={hdop:.1f}, VDOP={vdop:.1f}")

# For simulation, we'll continue rather than failing

# raise HealthCheckFailure("High DOP values")

elif max\_dop > 1.5:

logger.warning(f"High DOP: PDOP={pdop:.1f}, HDOP={hdop:.1f}, VDOP={vdop:.1f}")

else:

logger.info(f"DOP values OK: PDOP={pdop:.1f}, HDOP={hdop:.1f}, VDOP={vdop:.1f}")

self.health\_status['dop\_values'] = True

def check\_signal\_strength(self):

"""Verify signal strength for stable RTK fix."""

average\_snr = self.rtk\_rover.average\_snr

if average\_snr < 35:

logger.error(f"Weak signal: {average\_snr:.1f} dB-Hz")

# For simulation, we'll continue rather than failing

# raise HealthCheckFailure("Weak signal")

elif average\_snr < 45:

logger.warning(f"Weak signal: {average\_snr:.1f} dB-Hz")

else:

logger.info(f"Signal strength OK: {average\_snr:.1f} dB-Hz")

self.health\_status['signal\_strength'] = True

def check\_age\_of\_corrections(self):

"""Check age of RTK corrections for accuracy in milliseconds."""

# Updated to use milliseconds for more precise RTK correction age monitoring

aoc\_ms = self.rtk\_rover.age\_of\_corrections\_ms

# Define thresholds for warnings and errors

WARN\_MS = 100 # warn if older than 100 ms

ABORT\_MS = 200 # abort if older than 200 ms

if aoc\_ms > ABORT\_MS:

logger.error(f"Critical: corrections age is {aoc\_ms} ms – aborting to avoid float mode")

raise HealthCheckFailure(f"Corrections too old: {aoc\_ms} ms")

elif aoc\_ms > WARN\_MS:

logger.warning(f"Warning: corrections age is {aoc\_ms} ms – RTK accuracy may degrade")

else:

logger.info(f"Age of corrections OK: {aoc\_ms} ms")

self.health\_status['age\_of\_corrections'] = True

def check\_position\_validity(self):

"""Validate GPS position within expected range including farm boundaries."""

lat = self.rtk\_rover.latitude

lon = self.rtk\_rover.longitude

# Reference bounds for India

INDIA\_LAT\_MIN, INDIA\_LAT\_MAX = 5, 37

INDIA\_LON\_MIN, INDIA\_LON\_MAX = 60, 97

# Farm-specific bounds

FARM\_LAT\_MIN = self.rtk\_rover.LAT\_MIN

FARM\_LAT\_MAX = self.rtk\_rover.LAT\_MAX

FARM\_LON\_MIN = self.rtk\_rover.LON\_MIN

FARM\_LON\_MAX = self.rtk\_rover.LON\_MAX

# Check for zero coordinates or values outside allowed ranges

if (abs(lat) + abs(lon) == 0):

logger.error(f"Invalid position: zeros detected ({lat}, {lon})")

raise HealthCheckFailure("Zero coordinates detected")

# Check if within India's boundaries

if not (INDIA\_LAT\_MIN <= lat <= INDIA\_LAT\_MAX and INDIA\_LON\_MIN <= lon <= INDIA\_LON\_MAX):

logger.error(f"Invalid position: outside India bounds ({lat}, {lon})")

raise HealthCheckFailure("Coordinates outside valid range")

# Check if within farm boundaries

if not (FARM\_LAT\_MIN <= lat <= FARM\_LAT\_MAX and FARM\_LON\_MIN <= lon <= FARM\_LON\_MAX):

logger.error(f"Position outside farm boundaries: ({lat}, {lon})")

raise HealthCheckFailure("Position outside farm boundaries")

logger.info(f"Position valid: ({lat}, {lon}) - within farm boundaries")

self.health\_status['position\_validity'] = True

def check\_coordinate\_system(self):

"""Verify coordinate system conversion to UTM."""

if self.rtk\_rover.easting is None or self.rtk\_rover.northing is None:

logger.error("Coordinate conversion to UTM failed")

# For simulation, we'll continue rather than failing

# raise HealthCheckFailure("Coordinate error")

else:

logger.info(f"Coordinate system OK: UTM ({self.rtk\_rover.easting}, {self.rtk\_rover.northing})")

self.health\_status['coordinate\_system'] = True

def check\_multipath\_detection(self):

"""Detect multipath errors by checking SNR fluctuations."""

satellites = self.rtk\_rover.satellites\_data

snrs = [sat['snr'] for sat in satellites if sat['snr'] > 0]

if len(snrs) > 1:

snr\_std = np.std(snrs)

if snr\_std > 15:

logger.warning(f"Possible multipath: SNR std dev {snr\_std:.1f} dB-Hz")

# For simulation, we'll continue rather than failing

# raise HealthCheckFailure("Multipath error detected")

else:

logger.info(f"No multipath detected: SNR std dev {snr\_std:.1f} dB-Hz")

else:

logger.warning("Insufficient SNR data for multipath detection")

self.health\_status['multipath\_detection'] = True

def check\_gps\_sync\_time(self):

"""

Check GPS time synchronization with system clock with nanosecond precision.

Verifies time error and PPS jitter for precise timing needs.

"""

# Get time error in nanoseconds and PPS jitter

time\_error\_ns = self.rtk\_rover.time\_diff\_ns

pps\_jitter\_ns = self.rtk\_rover.pps\_jitter\_ns

# Define thresholds for warnings and errors

MAX\_TIME\_ERROR\_NS = 50 # 50 nanoseconds max error

MAX\_PPS\_JITTER\_NS = 20 # 20 nanoseconds max jitter

if time\_error\_ns is None:

logger.warning("Time error measurement not available")

return

# Check time error

if time\_error\_ns > MAX\_TIME\_ERROR\_NS:

logger.error(f"GPS time sync error: {time\_error\_ns} ns (exceeds {MAX\_TIME\_ERROR\_NS} ns)")

raise HealthCheckFailure(f"GPS time sync error: {time\_error\_ns} ns")

else:

logger.info(f"GPS time sync OK: {time\_error\_ns} ns")

# Check PPS jitter if available

if pps\_jitter\_ns is not None:

if pps\_jitter\_ns > MAX\_PPS\_JITTER\_NS:

logger.warning(f"PPS jitter high: {pps\_jitter\_ns} ns (exceeds {MAX\_PPS\_JITTER\_NS} ns)")

else:

logger.info(f"PPS jitter OK: {pps\_jitter\_ns} ns")

self.health\_status['gps\_sync\_time'] = True

def check\_receiver\_clock\_stability(self):

"""Check receiver clock stability by comparing GPS and system time."""

time\_diff = self.rtk\_rover.time\_diff

if time\_diff is None:

logger.warning("Time difference not available")

return

offset\_ms = time\_diff \* 1000

if offset\_ms > 200:

logger.error(f"Clock offset too high: {offset\_ms:.1f}ms")

# For simulation, we'll continue rather than failing

# raise HealthCheckFailure("Clock offset too high")

elif offset\_ms > 50:

logger.warning(f"Clock offset: {offset\_ms:.1f}ms")

else:

logger.info(f"Receiver clock stable: {offset\_ms:.1f}ms")

self.health\_status['receiver\_clock\_stability'] = True

def check\_power\_supply(self):

"""Check power supply by monitoring serial data flow."""

start\_time = time.time()

try:

data = self.rtk\_rover.ser.in\_waiting

if data == 0 and time.time() - start\_time > 5:

logger.error("No serial data received, possible power issue")

raise HealthCheckFailure("Power issue suspected")

else:

logger.info("Power supply OK: serial data received")

except serial.SerialException:

logger.error("Serial connection error, possible power issue")

raise HealthCheckFailure("Power issue suspected")

self.health\_status['power\_supply'] = True

def check\_firmware\_updates(self):

"""Check firmware status (manual verification required)."""

logger.warning("Please verify firmware is up to date via Emlid Flow app")

self.health\_status['firmware\_updates'] = True

def check\_battery\_level(self):

"""Check battery level (manual or hardware verification required)."""

logger.warning("Please ensure battery level is sufficient (>20%)")

self.health\_status['battery\_level'] = True

def check\_hardware\_status(self):

"""Check hardware status by monitoring serial connection."""

try:

data = self.rtk\_rover.ser.in\_waiting

if data == 0:

logger.error("No serial data, possible hardware issue")

raise HealthCheckFailure("Connection issue")

else:

logger.info("Hardware status OK: serial connection active")

except serial.SerialException:

logger.error("Serial connection error, possible hardware issue")

raise HealthCheckFailure("Connection issue")

self.health\_status['hardware\_status'] = True

def check\_constellation\_diversity(self):

"""Ensure multiple GNSS constellations for reliability."""

constellations = self.rtk\_rover.constellations

if len(constellations) < 2:

logger.warning(f"Low constellation diversity: {constellations}")

time.sleep(30)

raise HealthCheckFailure("Low constellation diversity")

else:

logger.info(f"Constellation diversity OK: {constellations}")

self.health\_status['constellation\_diversity'] = True

def check\_elevation\_mask(self):

"""Check satellite elevations to avoid multipath errors."""

min\_elevation = self.rtk\_rover.min\_elevation

if min\_elevation is None:

logger.warning("No elevation data")

return

if min\_elevation < 10:

logger.error(f"Low elevation: {min\_elevation}°")

raise HealthCheckFailure("Low satellite elevation")

elif min\_elevation < 15:

logger.warning(f"Low elevation: {min\_elevation}°")

else:

logger.info(f"Elevation mask OK: min elevation {min\_elevation}°")

self.health\_status['elevation\_mask'] = True

def check\_rtk\_initialization\_time(self, max\_time=120):

"""

Verify time to achieve RTK fixed status.

For optimal performance, RTK should initialize in under 20 seconds.

"""

# Start measuring initialization time

start\_time = time.time()

rtk\_fixed = False

init\_time = None

# Try for up to max\_time seconds to get RTK fixed

while time.time() - start\_time < max\_time:

self.rtk\_rover.read\_nmea\_data(num\_lines=10)

if self.rtk\_rover.fix\_quality == 4: # RTK Fixed

init\_time = time.time() - start\_time

rtk\_fixed = True

break

time.sleep(1)

# If we got a fix, check how long it took

if rtk\_fixed:

if init\_time <= 20:

logger.info(f"RTK fixed quickly in {init\_time:.1f}s - excellent performance")

elif init\_time <= 60:

logger.info(f"RTK fixed in {init\_time:.1f}s - acceptable performance")

else:

logger.warning(f"Slow RTK initialization: {init\_time:.1f}s - check base station visibility")

self.health\_status['rtk\_init\_time'] = True

return

else:

# Failed to get RTK fixed within max\_time

logger.error(f"RTK initialization failed - couldn't achieve fixed status in {max\_time}s")

raise HealthCheckFailure("RTK initialization timeout")

def check\_position\_stability(self, duration=60, interval=1, std\_threshold=0.02):

"""Check position stability over time."""

positions = []

start\_time = time.time()

while time.time() - start\_time < duration:

self.rtk\_rover.read\_nmea\_data(num\_lines=10)

if self.rtk\_rover.easting is not None and self.rtk\_rover.northing is not None:

positions.append((self.rtk\_rover.easting, self.rtk\_rover.northing))

time.sleep(interval)

if len(positions) < 2:

logger.warning("Insufficient position data")

return

eastings, northings = zip(\*positions)

std\_e = np.std(eastings)

std\_n = np.std(northings)

std\_total = np.sqrt(std\_e\*\*2 + std\_n\*\*2)

if std\_total > std\_threshold:

logger.warning(f"Unstable position: std dev {std\_total:.3f}m")

else:

logger.info(f"Position stable: std dev {std\_total:.3f}m")

self.health\_status['position\_stability'] = True

def run\_all\_checks(self, continue\_on\_failure=True, simulation\_mode=False):

"""Run all health checks sequentially.

Args:

continue\_on\_failure (bool): Continue testing even if a check fails

simulation\_mode (bool): If True, automatically pass all checks for simulation purposes

"""

logger.info(f"Starting rover RTK health check... {'(SIMULATION MODE)' if simulation\_mode else ''}")

failed\_checks = []

# If in simulation mode, automatically pass all checks

if simulation\_mode:

for key in self.health\_status:

self.health\_status[key] = True

logger.info("All health checks automatically passed for simulation mode")

return self.health\_status

try:

# First read data to populate rover fields

self.rover.read\_nmea\_data()

# Run checks

checks = [

('antenna\_placement', self.check\_antenna\_placement),

('rtk\_status', self.check\_rtk\_status),

('satellite\_count', self.check\_satellite\_count),

('dop\_values', self.check\_dop\_values),

('signal\_strength', self.check\_signal\_strength),

('age\_of\_corrections', self.check\_age\_of\_corrections),

('position\_validity', self.check\_position\_validity),

('coordinate\_system', self.check\_coordinate\_system),

('constellation\_diversity', self.check\_constellation\_diversity),

('elevation\_mask', self.check\_elevation\_mask),

('multipath\_detection', self.check\_multipath\_detection),

('gps\_sync\_time', self.check\_gps\_sync\_time),

('receiver\_clock\_stability', self.check\_receiver\_clock\_stability),

('power\_supply', self.check\_power\_supply),

('firmware\_updates', self.check\_firmware\_updates),

('battery\_level', self.check\_battery\_level),

('hardware\_status', self.check\_hardware\_status),

('rtk\_init\_time', self.check\_rtk\_initialization\_time),

('position\_stability', self.check\_position\_stability)

]

for check\_name, check\_func in checks:

try:

check\_func()

logger.info(f"Check {check\_name} passed.")

except HealthCheckFailure as e:

logger.error(f"Check {check\_name} failed: {e}")

failed\_checks.append(check\_name)

if not continue\_on\_failure:

raise

except Exception as e:

logger.error(f"Unexpected error during health checks: {e}")

if failed\_checks:

logger.warning(f"Failed checks: {', '.join(failed\_checks)}")

else:

logger.info("All health checks passed!")

return self.health\_status

def generate\_health\_report(self):

"""Generate a comprehensive health report."""

report = "=== ROVER HEALTH REPORT ===\n"

for system, status in self.health\_status.items():

status\_text = "PASS" if status else "FAIL"

report += f"{system.upper()}: {status\_text}\n"

all\_passed = all(self.health\_status.values())

report += "\nOVERALL STATUS: " + ("READY" if all\_passed else "NOT READY")

return report

if \_\_name\_\_ == "\_\_main\_\_":

# First create the RTK GPS rover instance

# Replace COM port as needed for your setup

rover = RTKGPSRover(port='COM3', baudrate=115200, log\_data=True)

# Then create the health checker using the rover

health\_checker = RoverHealthCheck(rover)

try:

# Run all health checks

health\_status = health\_checker.run\_all\_checks()

# Generate and print the health report

print(health\_checker.generate\_health\_report())

except HealthCheckFailure as e:

print(f"Health check failed: {e}")

finally:

# Close the serial connection when done

if rover.ser:

rover.ser.close()

print("Serial port closed")

1. sleep\_mode.py

"""

GNSS Rover Failsafe Module

--------------------------

This module implements the failsafe monitoring and recovery mechanisms for the rover

when dealing with GNSS-related issues. It can read data from an M2 GNSS receiver

or operate in simulation mode when the receiver is not present.

If failsafe conditions are not properly handled, the rover will enter sleep mode

to prevent dangerous operation.

"""

import enum

import time

import threading

import random

import math

import logging

import serial

import json

import os

from typing import Callable, Dict, List, Optional, Tuple, Union

from datetime import datetime

import pynmea2

# Configure logging

import logging

csv\_formatter = logging.Formatter('%(asctime)s,%(name)s,%(levelname)s,%(message)s')

logging.basicConfig(

level=logging.INFO,

format='%(asctime)s,%(name)s,%(levelname)s,%(message)s', # Applies only to basicConfig

handlers=[

logging.FileHandler(r"F:\GPS\task\_2\_waypoints\rover\_failsafe.csv"),

logging.StreamHandler()

]

)

# Apply formatter explicitly to ensure correct CSV output

for handler in logging.getLogger("rover\_failsafe").handlers:

handler.setFormatter(csv\_formatter)

logger = logging.getLogger("rover\_failsafe")

# Enum for GPS failsafe reasons

class GPSFailsafeReason(enum.Enum):

GPS\_STALE\_DATA = "GPS data stale (>300ms without NMEA fix)"

GPS\_CORRECTION\_STALE = "GPS corrections stale (>1000ms old sustained for >5000ms)"

GPS\_DATA\_LOSS = "Serial data loss (>300ms without NMEA on COM port)"

GPS\_FIX\_INSTABILITY = "Fix-status instability (≥3 FIX→FLOAT drops in 30000ms)"

GPS\_PERSISTENT\_DRIFT = "Persistent drift (≥6 drifts >3cm in 20000ms)"

GPS\_POSITION\_JUMP = "Position jump (>30cm or >0.5m/s implied speed)"

GPS\_HIGH\_DOP = "High DOP (PDOP >3.0 for >5000ms)"

GPS\_WEAK\_CONSTELLATION = "Weak constellation (<6 satellites for >5000ms)"

GPS\_MULTIPATH = "Signal multipath (C/N₀ drop >10dB-Hz for >5000ms)"

RTK\_FIX\_LOST = "RTK fix lost (>10000ms without RTK-FIX)"

INTERNET\_CONNECTION\_SLOW = "High NTRIP latency (avg correction age >1000ms for >10000ms)"

INTERNET\_CONNECTION\_LOST = "Complete internet loss (no corrections >5000ms AND no cell >10000ms)"

MODULE\_COMMUNICATION\_FAILURE = "Module communication failure"

UNKNOWN = "Unknown GPS failure"

PATH\_DEVIATION = "Path deviation (>5cm distance or >5° heading from planned path)"

LOW\_NTRIP\_DATA\_RATE = "Low NTRIP data rate (<2.4 kbps for >30000ms)"

# Enum for drift severity levels

class DriftSeverity(enum.Enum):

NONE = "No drift"

LOW = "Low drift (within tolerable limits)"

MEDIUM = "Medium drift (concerning)"

HIGH = "High drift (critical)"

# Enum for drift action responses

class DriftAction(enum.Enum):

CONTINUE = "Continue operation"

SLOW\_DOWN = "Reduce speed"

PAUSE = "Pause movement"

STOP = "Stop and wait for recovery"

SLEEP = "Enter sleep mode"

class FailsafeModule:

"""

Implements GNSS failsafe monitoring and handling based on the specified thresholds.

Can work with a real M2 GNSS receiver or in simulation mode.

"""

def \_\_init\_\_(self, port: str = "COM8", baud\_rate: int = 115200, simulation\_mode: bool = True):

self.snr\_values = {}

# Initialize timestamps

self.last\_gps\_update = time.time()

self.last\_internet\_check = time.time()

self.last\_module\_comm = time.time()

self.last\_position\_check = time.time()

# Serial connection parameters

self.port = port

self.baud\_rate = baud\_rate

self.serial\_conn = None

self.simulation\_mode = simulation\_mode

# GPS status

self.has\_fix = False

self.fix\_type = "NONE" # NONE, FLOAT, FIX

self.satellites = 0

self.hdop = 99.9

self.pdop = 99.9

self.vdop = 99.9

self.snr\_values = {} # Satellite PRN -> SNR value

self.position = (0.0, 0.0) # (latitude, longitude)

self.altitude = 0.0

# Internet and corrections status

self.internet\_connected = False

self.internet\_latency = 999.0

self.last\_correction = time.time() - 1000 # Start with old corrections

self.correction\_age = 999.0

self.ntrip\_data\_rate = 0.0 # Data rate in kbps

self.ntrip\_data\_rate\_check\_time = time.time()

# Failsafe monitoring

self.monitoring\_thread = None

self.stop\_thread = False

self.active\_failsafe = None

self.in\_recovery = False

self.in\_sleep\_mode = False

self.sleep\_start\_time = 0

self.sleep\_duration = 0

# Statistics for monitoring

self.fix\_drops = [] # List of timestamps when FIX->FLOAT drops occurred

self.position\_history = [] # List of (timestamp, position) tuples

self.drift\_events = [] # List of (timestamp, drift\_size) tuples

# Callbacks

self.on\_failsafe\_triggered = None

self.on\_recovery\_attempt = None

self.on\_rover\_wakeup = None

# Dependencies

self.safety\_module = None

self.in\_failsafe\_mode = False

# Add path tracking fields

self.planned\_path = [] # List of waypoints

self.planned\_heading = 0.0 # Planned heading in degrees

self.current\_heading = 0.0 # Current heading in degrees

self.path\_deviation\_events = [] # List of (timestamp, distance\_dev, heading\_dev) tuples

self.last\_waypoint = None # Last successfully reached waypoint

# Configure simulation parameters for testing

self.simulation\_params = {

"path\_deviation\_prob": 0.01, # Probability of path deviation

"stale\_data\_prob": 0.005, # Probability of stale data per check

"correction\_stale\_prob": 0.005, # Probability of stale corrections

"data\_loss\_prob": 0.005, # Probability of serial data loss

"fix\_drop\_prob": 0.01, # Probability of FIX->FLOAT drop

"drift\_prob": 0.01, # Probability of position drift

"jump\_prob": 0.005, # Probability of position jump

"high\_dop\_prob": 0.01, # Probability of high DOP

"weak\_constellation\_prob": 0.01, # Probability of weak constellation

"multipath\_prob": 0.01, # Probability of signal multipath

"rtk\_loss\_prob": 0.01, # Probability of RTK fix loss

"internet\_latency\_prob": 0.01, # Probability of high internet latency

"internet\_loss\_prob": 0.005, # Probability of complete internet loss

"low\_ntrip\_data\_rate\_prob": 0.005, # Probability of low NTRIP data rate

}

# Updated thresholds based on requirements

self.thresholds = {

"gps\_stale\_data": 0.3, # 300ms without NMEA fix

"gps\_correction\_stale\_age": 0.3, # 300ms correction age

"gps\_correction\_stale\_duration": 2.0, # 5000ms duration

"gps\_data\_loss": 0.3, # 300ms without NMEA on COM port

"fix\_instability\_count": 3, # 3 FIX->FLOAT drops

"fix\_instability\_window": 30.0, # 30000ms window

"persistent\_drift\_count": 6, # 6 drifts

"persistent\_drift\_size": 0.03, # 3cm drift

"persistent\_drift\_window": 20.0, # 20000ms window

"position\_jump\_distance": 0.3, # 30cm jump

"position\_jump\_speed": 0.5, # 0.5m/s implied speed

"high\_dop\_threshold": 3.0, # PDOP > 3.0

"high\_dop\_duration": 5.0, # 5000ms duration

"weak\_constellation\_count": 6, # <6 satellites

"weak\_constellation\_duration": 5.0, # 5000ms duration

"multipath\_drop": 10.0, # 10dB-Hz drop

"multipath\_duration": 5.0, # 5000ms duration

"rtk\_fix\_lost\_duration": 10.0, # 10000ms without RTK-FIX

"ntrip\_latency\_threshold": 1.0, # 1000ms correction age

"ntrip\_latency\_duration": 10.0, # 10000ms duration

"internet\_loss\_corrections": 5.0, # 5000ms without corrections

"internet\_loss\_cellular": 10.0, # 10000ms without cellular

"path\_deviation\_distance": 0.05, # 5cm distance deviation

"path\_deviation\_heading": 5.0, # 5° heading deviation

"low\_ntrip\_data\_rate": 2.4, # 2.4 kbps

"low\_ntrip\_data\_rate\_duration": 30.0, # 30000ms duration

}

# Recovery times based on requirements

self.recovery\_times = {

GPSFailsafeReason.GPS\_STALE\_DATA: 10, # 10s to reconnect GNSS

GPSFailsafeReason.GPS\_CORRECTION\_STALE: 20, # 20s to reconnect NTRIP

GPSFailsafeReason.GPS\_DATA\_LOSS: 15, # 15s to reconnect GNSS

GPSFailsafeReason.GPS\_FIX\_INSTABILITY: 30, # 30s to monitor

GPSFailsafeReason.GPS\_PERSISTENT\_DRIFT: 20, # 20s to monitor

GPSFailsafeReason.GPS\_POSITION\_JUMP: 10, # 10s to stop and check

GPSFailsafeReason.GPS\_HIGH\_DOP: 15, # 15s to wait

GPSFailsafeReason.GPS\_WEAK\_CONSTELLATION: 20, # 20s to wait

GPSFailsafeReason.GPS\_MULTIPATH: 5, # 5s before sleep

GPSFailsafeReason.RTK\_FIX\_LOST: 30, # 30s to monitor

GPSFailsafeReason.INTERNET\_CONNECTION\_SLOW: 20, # 20s to reconnect

GPSFailsafeReason.INTERNET\_CONNECTION\_LOST: 30, # 30s to reconnect

GPSFailsafeReason.MODULE\_COMMUNICATION\_FAILURE: 15, # 15s to reconnect

GPSFailsafeReason.PATH\_DEVIATION: 20, # 20s to adjust path

GPSFailsafeReason.LOW\_NTRIP\_DATA\_RATE: 30, # 30s to reconnect NTRIP

GPSFailsafeReason.UNKNOWN: 10 # 10s default

}

# Sleep duration in seconds (5 minutes)

self.default\_sleep\_duration = 300

logger.info("FailsafeModule initialized in %s mode",

"simulation" if simulation\_mode else "hardware")

def set\_safety\_module(self, safety\_module):

"""Set the safety module reference"""

self.safety\_module = safety\_module

def set\_callbacks(self, on\_failsafe\_triggered: Callable = None,

on\_recovery\_attempt: Callable = None,

on\_rover\_wakeup: Callable = None):

"""Set callback functions"""

self.on\_failsafe\_triggered = on\_failsafe\_triggered

self.on\_recovery\_attempt = on\_recovery\_attempt

self.on\_rover\_wakeup = on\_rover\_wakeup

def connect\_to\_gnss(self) -> bool:

"""

Connect to M2 GNSS receiver through serial port

Returns True if connection successful, False otherwise

"""

if self.simulation\_mode:

logger.info("Running in simulation mode, no physical GNSS connection required")

return True

try:

self.serial\_conn = serial.Serial(self.port, self.baud\_rate, timeout=1)

logger.info(f"Connected to GNSS receiver on {self.port} at {self.baud\_rate} baud")

return True

except Exception as e:

logger.error(f"Failed to connect to GNSS receiver: {e}")

return False

def read\_gnss\_data(self) -> Dict:

"""

Read data from M2 GNSS receiver

Returns a dictionary with parsed GNSS data

"""

if self.simulation\_mode:

# Generate simulated GNSS data

return self.\_generate\_simulated\_gnss\_data()

if not self.serial\_conn or not self.serial\_conn.is\_open:

if not self.connect\_to\_gnss():

return {}

try:

# Read data from serial port

line = self.serial\_conn.readline().decode('ascii', errors='replace').strip()

if not line:

return {}

# Parse NMEA sentence

try:

msg = pynmea2.parse(line)

data = self.\_parse\_nmea\_message(msg)

self.last\_gps\_update = time.time()

return data

except pynmea2.ParseError:

return {}

except Exception as e:

logger.error(f"Error reading GNSS data: {e}")

return {}

def \_parse\_nmea\_message(self, msg) -> Dict:

"""Parse different types of NMEA messages and extract relevant data"""

data = {}

try:

if isinstance(msg, pynmea2.GGA):

# Global Positioning System Fix Data

data["fix\_type"] = int(msg.gps\_qual)

data["satellites"] = int(msg.num\_sats) if msg.num\_sats else 0

data["hdop"] = float(msg.horizontal\_dil) if msg.horizontal\_dil else 99.9

data["position"] = (float(msg.latitude), float(msg.longitude))

data["altitude"] = float(msg.altitude) if msg.altitude else 0.0

data["correction\_age"] = float(msg.age\_gps\_data) if msg.age\_gps\_data else 999.0

# Update fix quality based on GGA quality indicator

if data["fix\_type"] == 4:

data["fix\_quality"] = "FIX" # RTK fixed solution

elif data["fix\_type"] == 5:

data["fix\_quality"] = "FLOAT" # RTK float solution

elif data["fix\_type"] > 0:

data["fix\_quality"] = "GPS" # Standard GPS fix

else:

data["fix\_quality"] = "NONE" # No fix

elif isinstance(msg, pynmea2.GSA):

# GPS DOP and active satellites

data["pdop"] = float(msg.pdop) if msg.pdop else 99.9

data["hdop"] = float(msg.hdop) if msg.hdop else 99.9

data["vdop"] = float(msg.vdop) if msg.vdop else 99.9

data["fix\_type\_3d"] = int(msg.mode\_fix\_type) if msg.mode\_fix\_type else 0

elif isinstance(msg, pynmea2.GSV):

# Satellites in view

if msg.msg\_num == 1: # First message in sequence

self.snr\_values = {} # Reset SNR values

current\_time = time.time()

for sat\_index in range(4): # Each GSV message contains up to 4 satellites

sat\_num\_attr = f"sv\_prn\_{sat\_index + 1}"

snr\_attr = f"snr\_{sat\_index + 1}"

if hasattr(msg, sat\_num\_attr) and hasattr(msg, snr\_attr):

sat\_num = getattr(msg, sat\_num\_attr)

snr = getattr(msg, snr\_attr)

if sat\_num and snr:

if sat\_num not in self.snr\_values:

self.snr\_values[sat\_num] = []

self.snr\_values[sat\_num].append((current\_time, float(snr)))

data["snr\_values"] = self.snr\_values.copy()

elif isinstance(msg, pynmea2.VTG):

# Track made good and ground speed

data["speed"] = float(msg.spd\_over\_grnd\_kmph) / 3.6 # km/h → m/s

data["track"] = float(msg.true\_track) if msg.true\_track else 0.0

elif isinstance(msg, pynmea2.RMC):

# Recommended minimum specific GPS/Transit data

if msg.status == 'A': # A=active, V=void

data["position"] = (float(msg.latitude), float(msg.longitude))

data["speed"] = float(msg.spd\_over\_grnd) \* 0.514444 # knots → m/s

data["track"] = float(msg.true\_course) if msg.true\_course else 0.0

data["timestamp"] = (

msg.datetime.timestamp() if msg.datetime else time.time()

)

except Exception as e:

logger.error(f"Error parsing NMEA message: {e}")

return data

def \_generate\_simulated\_gnss\_data(self) -> Dict:

"""Generate simulated GNSS data for testing"""

current\_time = time.time()

data = {}

# Base values

data["timestamp"] = current\_time

# Simulate realistic values with occasional failures based on simulation parameters

# Add heading information

if hasattr(self, 'current\_heading') and not hasattr(self, 'sim\_heading'):

self.sim\_heading = random.uniform(0, 360)

# Simulate heading

if self.active\_failsafe == GPSFailsafeReason.PATH\_DEVIATION:

# Introduce a significant heading deviation

planned\_heading = getattr(self, 'planned\_heading', 0)

heading\_deviation = random.uniform(6, 20) # More than 5 degrees

if random.random() < 0.5:

data["heading"] = (planned\_heading + heading\_deviation) % 360

else:

data["heading"] = (planned\_heading - heading\_deviation) % 360

else:

# Normal small heading variations

if hasattr(self, 'sim\_heading'):

data["heading"] = (self.sim\_heading + random.uniform(-3, 3)) % 360

self.sim\_heading = data["heading"] # Update for next time

else:

data["heading"] = random.uniform(0, 360)

# Fix type

if self.active\_failsafe == GPSFailsafeReason.RTK\_FIX\_LOST:

data["fix\_quality"] = "FLOAT"

elif self.active\_failsafe == GPSFailsafeReason.GPS\_FIX\_INSTABILITY:

# Occasionally switch between FIX and FLOAT

data["fix\_quality"] = "FLOAT" if random.random() < 0.5 else "FIX"

else:

# Normally provide RTK fix with occasional drops based on probability

if random.random() < self.simulation\_params["fix\_drop\_prob"]:

data["fix\_quality"] = "FLOAT"

else:

data["fix\_quality"] = "FIX"

# Satellites

if self.active\_failsafe == GPSFailsafeReason.GPS\_WEAK\_CONSTELLATION:

data["satellites"] = random.randint(3, 5) # Below threshold of 6

else:

data["satellites"] = random.randint(8, 16) # Normal range

# DOP values

if self.active\_failsafe == GPSFailsafeReason.GPS\_HIGH\_DOP:

data["pdop"] = random.uniform(3.5, 6.0) # Above threshold of 3.0

else:

data["pdop"] = random.uniform(1.2, 2.5) # Normal range

data["hdop"] = data["pdop"] \* random.uniform(0.6, 0.8)

data["vdop"] = data["pdop"] \* random.uniform(0.7, 0.9)

# SNR values

snr\_values = {}

for i in range(1, data["satellites"] + 1):

base\_snr = random.uniform(35, 50) # Normal range

if self.active\_failsafe == GPSFailsafeReason.GPS\_MULTIPATH and random.random() < 0.7:

base\_snr -= random.uniform(12, 20) # Drop by more than 10 dB-Hz

snr\_values[f"G{i}"] = base\_snr

data["snr\_values"] = snr\_values

# Position

if not hasattr(self, 'sim\_position') or not self.sim\_position:

self.sim\_position = (random.uniform(40.0, 41.0), random.uniform(-74.0, -73.0))

# Handle position drift and jumps

if self.active\_failsafe == GPSFailsafeReason.GPS\_POSITION\_JUMP:

# Large jump

jump\_size = random.uniform(0.4, 1.0)

jump\_angle = random.uniform(0, math.pi \* 2)

delta\_lat = jump\_size \* math.cos(jump\_angle) / 111111

delta\_lon = jump\_size \* math.sin(jump\_angle) / (111111 \* math.cos(math.radians(self.sim\_position[0])))

data["position"] = (self.sim\_position[0] + delta\_lat, self.sim\_position[1] + delta\_lon)

elif self.active\_failsafe == GPSFailsafeReason.GPS\_PERSISTENT\_DRIFT:

# Persistent small drifts

drift\_size = random.uniform(0.03, 0.10) # 3-10 cm

drift\_angle = random.uniform(0, math.pi \* 2)

delta\_lat = drift\_size \* math.cos(drift\_angle) / 111111

delta\_lon = drift\_size \* math.sin(drift\_angle) / (111111 \* math.cos(math.radians(self.sim\_position[0])))

data["position"] = (self.sim\_position[0] + delta\_lat, self.sim\_position[1] + delta\_lon)

else:

# Normal small variations

delta\_lat = random.uniform(-0.01, 0.01) / 111111 # ~1cm

delta\_lon = random.uniform(-0.01, 0.01) / (111111 \* math.cos(math.radians(self.sim\_position[0])))

data["position"] = (self.sim\_position[0] + delta\_lat, self.sim\_position[1] + delta\_lon)

self.sim\_position = data["position"] # Update simulated position for next time

# Altitude

data["altitude"] = random.uniform(100.0, 101.0)

# Speed and heading

data["speed"] = random.uniform(0.1, 0.5) # m/s

data["track"] = random.uniform(0, 360) # degrees

# Correction age

if self.active\_failsafe == GPSFailsafeReason.GPS\_CORRECTION\_STALE:

data["correction\_age"] = random.uniform(1.5, 10.0) # Above threshold (1000ms)

elif self.active\_failsafe == GPSFailsafeReason.INTERNET\_CONNECTION\_SLOW:

data["correction\_age"] = random.uniform(1.5, 6.0) # Above threshold (1000ms)

else:

data["correction\_age"] = random.uniform(0.1, 0.8) # Normal range

# NTRIP data rate

if self.active\_failsafe == GPSFailsafeReason.LOW\_NTRIP\_DATA\_RATE:

data["ntrip\_data\_rate"] = random.uniform(0.5, 2.0) # Below threshold (2.4 kbps)

else:

data["ntrip\_data\_rate"] = random.uniform(3.0, 10.0) # Normal range

return data

def update\_path\_info(self, planned\_path: List[Tuple[float, float]] = None,

planned\_heading: float = None,

current\_heading: float = None,

last\_waypoint: Tuple[float, float] = None) -> None:

"""Update the current path planning information"""

if planned\_path is not None:

self.planned\_path = planned\_path

if planned\_heading is not None:

self.planned\_heading = planned\_heading

if current\_heading is not None:

self.current\_heading = current\_heading

if last\_waypoint is not None:

self.last\_waypoint = last\_waypoint

def update\_gps\_status(self, has\_fix: bool = None, fix\_type: str = None,

satellites: int = None, hdop: float = None,

pdop: float = None, position: Tuple[float, float] = None,

altitude: float = None) -> None:

"""Update the current GPS status with new values"""

current\_time = time.time()

# Only update values that are provided

if has\_fix is not None:

self.has\_fix = has\_fix

if fix\_type is not None:

# If changing from FIX to FLOAT, record as a drop event

if self.fix\_type == "FIX" and fix\_type == "FLOAT":

self.fix\_drops.append(current\_time)

# Clean up old drops (more than 30 seconds old)

self.fix\_drops = [t for t in self.fix\_drops if current\_time - t <= self.thresholds["fix\_instability\_window"]]

self.fix\_type = fix\_type

if satellites is not None:

self.satellites = satellites

if hdop is not None:

self.hdop = hdop

if pdop is not None:

self.pdop = pdop

if position is not None:

# Record position for drift/jump detection

self.position\_history.append((current\_time, position))

# Clean up old positions (more than 30 seconds old)

self.position\_history = [(t, p) for t, p in self.position\_history

if current\_time - t <= 30]

# Check for jumps or drifts if we have previous positions

if len(self.position\_history) > 1:

prev\_time, prev\_pos = self.position\_history[-2]

dist = self.\_calculate\_distance(prev\_pos, position)

time\_diff = current\_time - prev\_time

# Only check if time difference is reasonable

if time\_diff > 0:

# Detect jumps (>30cm or implied speed >0.5m/s)

if dist > self.thresholds["position\_jump\_distance"] or (dist / time\_diff) > self.thresholds["position\_jump\_speed"]:

logger.warning(f"Position jump detected: {dist:.2f}m in {time\_diff:.2f}s")

# Detect drifts (>3cm)

if dist > self.thresholds["persistent\_drift\_size"]:

self.drift\_events.append((current\_time, dist))

# Clean up old drift events (more than 20 seconds old)

self.drift\_events = [(t, d) for t, d in self.drift\_events

if current\_time - t <= self.thresholds["persistent\_drift\_window"]]

self.position = position

if altitude is not None:

self.altitude = altitude

self.last\_gps\_update = current\_time

def update\_internet\_status(self, connected: bool = None, latency: float = None) -> None:

"""Update the current internet connection status"""

current\_time = time.time()

if connected is not None:

self.internet\_connected = connected

if latency is not None:

self.internet\_latency = latency

self.last\_internet\_check = current\_time

def update\_correction\_status(self, age: float = None) -> None:

"""Update the current correction status"""

current\_time = time.time()

if age is not None:

self.correction\_age = age

self.last\_correction = current\_time - age

def update\_ntrip\_data\_rate(self, data\_rate: float = None) -> None:

"""Update the current NTRIP data rate in kbps"""

current\_time = time.time()

if data\_rate is not None:

self.ntrip\_data\_rate = data\_rate

self.ntrip\_data\_rate\_check\_time = current\_time

def update\_module\_communication(self) -> None:

"""Update the timestamp for last successful module communication"""

self.last\_module\_comm = time.time()

def \_calculate\_distance(self, pos1: Tuple[float, float], pos2: Tuple[float, float]) -> float:

"""

Calculate the distance between two positions in meters

Uses Haversine formula for accurate earth distance

"""

# For simplicity in simulation, use a flat-earth approximation

# 1 degree latitude = ~111,111 meters

# 1 degree longitude = ~111,111 \* cos(latitude) meters

lat1, lon1 = pos1

lat2, lon2 = pos2

# Convert to radians

lat1\_rad = math.radians(lat1)

lat2\_rad = math.radians(lat2)

lon1\_rad = math.radians(lon1)

lon2\_rad = math.radians(lon2)

# Haversine formula

dlon = lon2\_rad - lon1\_rad

dlat = lat2\_rad - lat1\_rad

a = (math.sin(dlat/2)\*\*2) + math.cos(lat1\_rad) \* math.cos(lat2\_rad) \* (math.sin(dlon/2)\*\*2)

c = 2 \* math.atan2(math.sqrt(a), math.sqrt(1-a))

distance = 6371000 \* c # Earth radius in meters

return distance

def get\_failsafe\_reason(self) -> Optional[GPSFailsafeReason]:

"""

Check all failsafe conditions and return the reason if any is triggered

Returns None if all is well

"""

current\_time = time.time()

# 1. GPS data stale (>300ms without any NMEA fix)

if current\_time - self.last\_gps\_update > self.thresholds["gps\_stale\_data"]:

return GPSFailsafeReason.GPS\_STALE\_DATA

# 2. GPS corrections stale (>1000ms old sustained for >5000ms)

if self.correction\_age > self.thresholds["gps\_correction\_stale\_age"] and current\_time - self.last\_correction > self.thresholds["gps\_correction\_stale\_duration"]:

return GPSFailsafeReason.GPS\_CORRECTION\_STALE

# 3. Serial data loss (>300ms without NMEA on COM port)

if current\_time - self.last\_module\_comm > self.thresholds["gps\_data\_loss"]:

return GPSFailsafeReason.GPS\_DATA\_LOSS

# 4. Fix-status instability (≥3 FIX→FLOAT drops in 30000ms)

if len(self.fix\_drops) >= self.thresholds["fix\_instability\_count"]:

return GPSFailsafeReason.GPS\_FIX\_INSTABILITY

# 5. Persistent drift (≥6 drifts >3cm in 20000ms)

if len(self.drift\_events) >= self.thresholds["persistent\_drift\_count"]:

return GPSFailsafeReason.GPS\_PERSISTENT\_DRIFT

# 6. Position jump (>30cm or implied speed >0.5m/s)

# This is checked during position updates

if len(self.position\_history) >= 2:

last\_time, last\_pos = self.position\_history[-1]

prev\_time, prev\_pos = self.position\_history[-2]

time\_diff = last\_time - prev\_time

if time\_diff > 0:

dist = self.\_calculate\_distance(prev\_pos, last\_pos)

if dist > self.thresholds["position\_jump\_distance"] or (dist / time\_diff) > self.thresholds["position\_jump\_speed"]:

return GPSFailsafeReason.GPS\_POSITION\_JUMP

# 7. High DOP (PDOP >3.0 for >5000ms)

if self.pdop > self.thresholds["high\_dop\_threshold"] and current\_time - self.last\_position\_check > self.thresholds["high\_dop\_duration"]:

return GPSFailsafeReason.GPS\_HIGH\_DOP

# 8. Weak constellation (<6 satellites for >5000ms)

if self.satellites < self.thresholds["weak\_constellation\_count"] and current\_time - self.last\_position\_check > self.thresholds["weak\_constellation\_duration"]:

return GPSFailsafeReason.GPS\_WEAK\_CONSTELLATION

# 9. Signal multipath (C/N₀ drop >10dB-Hz for >5000ms)

# Check for multipath failsafe

for prn in list(self.snr\_values.keys()):

# Clean up entries older than 60 seconds to limit memory usage

self.snr\_values[prn] = [(t, snr) for t, snr in self.snr\_values[prn] if current\_time - t <= 60]

if not self.snr\_values[prn]: # Remove empty lists

del self.snr\_values[prn]

continue

snr\_history = self.snr\_values[prn]

if len(snr\_history) < 2: # Need sufficient history to compare

continue

# Calculate baseline SNR (average over last 30 seconds)

baseline\_snrs = [snr for t, snr in snr\_history if current\_time - t <= 30]

if not baseline\_snrs:

continue

baseline\_avg = sum(baseline\_snrs) / len(baseline\_snrs)

# Get recent SNR values (last 5 seconds)

recent\_snrs = [snr for t, snr in snr\_history if current\_time - t <= self.thresholds["multipath\_duration"]]

# Ensure enough data points (e.g., at least 3) and check if all are below baseline - 10

if len(recent\_snrs) >= 3 and all(snr < baseline\_avg - self.thresholds["multipath\_drop"] for snr in recent\_snrs):

return GPSFailsafeReason.GPS\_MULTIPATH

# 10. RTK fix lost (>10000ms without RTK-FIX)

if self.fix\_type != "FIX" and current\_time - self.last\_position\_check > self.thresholds["rtk\_fix\_lost\_duration"]:

return GPSFailsafeReason.RTK\_FIX\_LOST

# 11. High NTRIP latency (avg correction age >1000ms for >10000ms)

if self.correction\_age > self.thresholds["ntrip\_latency\_threshold"] and current\_time - self.last\_internet\_check > self.thresholds["ntrip\_latency\_duration"]:

return GPSFailsafeReason.INTERNET\_CONNECTION\_SLOW

# 12. Complete internet loss (no corrections >5000ms AND no cell >10000ms)

if not self.internet\_connected and current\_time - self.last\_correction > self.thresholds["internet\_loss\_corrections"] and current\_time - self.last\_internet\_check > self.thresholds["internet\_loss\_cellular"]:

return GPSFailsafeReason.INTERNET\_CONNECTION\_LOST

# 13. Low NTRIP data rate (<2.4 kbps for >30000ms)

if self.ntrip\_data\_rate < self.thresholds["low\_ntrip\_data\_rate"] and current\_time - self.ntrip\_data\_rate\_check\_time > self.thresholds["low\_ntrip\_data\_rate\_duration"]:

return GPSFailsafeReason.LOW\_NTRIP\_DATA\_RATE

# 14. Check for path deviation (if we have a planned path)

if self.planned\_path and len(self.planned\_path) > 0:

deviation\_detected, distance\_dev, heading\_dev = self.check\_path\_deviation(

self.position, self.current\_heading, self.planned\_path, self.planned\_heading)

if deviation\_detected:

# Record this deviation event

self.path\_deviation\_events.append((current\_time, distance\_dev, heading\_dev))

# Clean up old events (more than 15 seconds old)

self.path\_deviation\_events = [(t, d, h) for t, d, h in self.path\_deviation\_events

if current\_time - t <= 15]

# If we have multiple deviation events in a short time, trigger failsafe

if len(self.path\_deviation\_events) >= 3: # At least 3 events

return GPSFailsafeReason.PATH\_DEVIATION

# No failsafe triggered

return None

def start\_monitoring(self) -> None:

"""Start the failsafe monitoring thread"""

if self.monitoring\_thread and self.monitoring\_thread.is\_alive():

logger.warning("Monitoring thread is already running")

return

self.stop\_thread = False

self.monitoring\_thread = threading.Thread(target=self.\_monitoring\_loop, daemon=True)

self.monitoring\_thread.start()

logger.info("Failsafe monitoring started")

def stop\_monitoring(self) -> None:

"""Stop the failsafe monitoring thread"""

self.stop\_thread = True

if self.monitoring\_thread:

self.monitoring\_thread.join(timeout=2)

logger.info("Failsafe monitoring stopped")

def \_monitoring\_loop(self) -> None:

"""Main monitoring loop that runs in a separate thread"""

check\_interval = 0.2 # Check every 200ms

last\_record\_time = time.time()

record\_interval = 5.0 # Record status every 5 seconds

while not self.stop\_thread:

try:

current\_time = time.time()

# Inject simulated failures occasionally in simulation mode

if self.simulation\_mode and not self.active\_failsafe:

self.\_inject\_simulated\_failure()

# Read from actual GNSS receiver if available and not in simulation

if not self.simulation\_mode and (current\_time - self.last\_gps\_update > 1):

gnss\_data = self.read\_gnss\_data()

if gnss\_data:

# Update status based on real data

if "fix\_quality" in gnss\_data:

self.update\_gps\_status(

has\_fix=(gnss\_data["fix\_quality"] != "NONE"),

fix\_type=gnss\_data["fix\_quality"],

satellites=gnss\_data.get("satellites", self.satellites),

hdop=gnss\_data.get("hdop", self.hdop),

pdop=gnss\_data.get("pdop", self.pdop),

position=gnss\_data.get("position", self.position),

altitude=gnss\_data.get("altitude", self.altitude)

)

if "correction\_age" in gnss\_data:

self.update\_correction\_status(age=gnss\_data["correction\_age"])

if "ntrip\_data\_rate" in gnss\_data:

self.update\_ntrip\_data\_rate(data\_rate=gnss\_data["ntrip\_data\_rate"])

self.update\_module\_communication()

# Check for failsafe conditions

if not self.in\_recovery:

reason = self.get\_failsafe\_reason()

if reason:

self.\_handle\_failsafe(reason)

# Log status periodically

if current\_time - last\_record\_time >= record\_interval:

self.\_log\_status()

last\_record\_time = current\_time

# Update last position check time

self.last\_position\_check = current\_time

# Sleep for a bit

time.sleep(check\_interval)

except Exception as e:

logger.error(f"Error in monitoring loop: {e}")

time.sleep(1) # Wait a bit longer on error

def \_inject\_simulated\_failure(self) -> None:

"""Inject a simulated failure for testing purposes"""

# Only run this occasionally

if random.random() > 0.01: # 1% chance per check (5 times per second)

return

# Choose a random failure type weighted by probability

failure\_types = [

(GPSFailsafeReason.GPS\_STALE\_DATA, self.simulation\_params["stale\_data\_prob"]),

(GPSFailsafeReason.GPS\_CORRECTION\_STALE, self.simulation\_params["correction\_stale\_prob"]),

(GPSFailsafeReason.GPS\_DATA\_LOSS, self.simulation\_params["data\_loss\_prob"]),

(GPSFailsafeReason.GPS\_FIX\_INSTABILITY, self.simulation\_params["fix\_drop\_prob"]),

(GPSFailsafeReason.GPS\_PERSISTENT\_DRIFT, self.simulation\_params["drift\_prob"]),

(GPSFailsafeReason.GPS\_POSITION\_JUMP, self.simulation\_params["jump\_prob"]),

(GPSFailsafeReason.GPS\_HIGH\_DOP, self.simulation\_params["high\_dop\_prob"]),

(GPSFailsafeReason.GPS\_WEAK\_CONSTELLATION, self.simulation\_params["weak\_constellation\_prob"]),

(GPSFailsafeReason.GPS\_MULTIPATH, self.simulation\_params["multipath\_prob"]),

(GPSFailsafeReason.RTK\_FIX\_LOST, self.simulation\_params["rtk\_loss\_prob"]),

(GPSFailsafeReason.INTERNET\_CONNECTION\_SLOW, self.simulation\_params["internet\_latency\_prob"]),

(GPSFailsafeReason.INTERNET\_CONNECTION\_LOST, self.simulation\_params["internet\_loss\_prob"]),

(GPSFailsafeReason.PATH\_DEVIATION, self.simulation\_params["path\_deviation\_prob"]),

(GPSFailsafeReason.LOW\_NTRIP\_DATA\_RATE, self.simulation\_params["low\_ntrip\_data\_rate\_prob"]),

]

# Calculate total probability for normalization

total\_prob = sum(prob for \_, prob in failure\_types)

if total\_prob <= 0:

return

# Normalize probabilities

normalized\_probs = [prob/total\_prob for \_, prob in failure\_types]

# Choose a failure type based on probability

chosen\_failure = random.choices(

[failure for failure, \_ in failure\_types],

weights=normalized\_probs,

k=1

)[0]

# Set the active failsafe

# Set the active failsafe

logger.info(f"Injecting simulated failure: {chosen\_failure.value}")

self.active\_failsafe = chosen\_failure

# Make the failure persist for a realistic time

# It will be cleared when recovery is successful

# The \_handle\_failsafe method will be called on the next monitoring cycle

def \_handle\_failsafe(self, reason: GPSFailsafeReason) -> None:

"""Handle a detected failsafe condition"""

if self.active\_failsafe == reason:

# Already handling this failsafe

return

logger.warning(f"GNSS Failsafe triggered: {reason.value}")

self.active\_failsafe = reason

# Call the callback if registered

if self.on\_failsafe\_triggered:

try:

self.on\_failsafe\_triggered(reason)

except Exception as e:

logger.error(f"Error in failsafe callback: {e}")

# Start recovery process

self.start\_recovery()

def start\_recovery(self) -> None:

"""Start the failsafe recovery process"""

if self.in\_recovery:

logger.warning("Already in recovery mode, ignoring new recovery request")

return

self.in\_recovery = True

logger.info(f"Starting recovery for: {self.active\_failsafe.value}")

# Call the recovery callback if registered

if self.on\_recovery\_attempt:

try:

self.on\_recovery\_attempt(self.active\_failsafe)

except Exception as e:

logger.error(f"Error in recovery callback: {e}")

# Start recovery thread

recovery\_thread = threading.Thread(target=self.\_recovery\_process, daemon=True)

recovery\_thread.start()

def \_recovery\_process(self) -> None:

"""

The recovery process runs in a separate thread.

It tries different recovery strategies depending on the failsafe reason.

"""

try:

# Get recovery time for this failure type

recovery\_time = self.recovery\_times.get(self.active\_failsafe, 15)

# Strategy depends on the failure type

if self.active\_failsafe == GPSFailsafeReason.GPS\_STALE\_DATA:

self.\_attempt\_reconnect\_gnss()

elif self.active\_failsafe == GPSFailsafeReason.GPS\_DATA\_LOSS:

self.\_attempt\_reconnect\_gnss()

elif self.active\_failsafe == GPSFailsafeReason.MODULE\_COMMUNICATION\_FAILURE:

self.\_attempt\_reconnect\_gnss()

elif self.active\_failsafe in [GPSFailsafeReason.INTERNET\_CONNECTION\_SLOW,

GPSFailsafeReason.INTERNET\_CONNECTION\_LOST]:

self.\_attempt\_reconnect\_internet()

elif self.active\_failsafe in [GPSFailsafeReason.GPS\_CORRECTION\_STALE,

GPSFailsafeReason.LOW\_NTRIP\_DATA\_RATE]:

self.\_attempt\_reconnect\_ntrip()

elif self.active\_failsafe == GPSFailsafeReason.PATH\_DEVIATION:

self.\_attempt\_path\_recovery()

# Wait for conditions to improve up to recovery\_time seconds

start\_time = time.time()

while time.time() - start\_time < recovery\_time:

# Check if the failsafe condition is still active

current\_reason = self.get\_failsafe\_reason()

if current\_reason is None or current\_reason != self.active\_failsafe:

# Condition is resolved or changed

logger.info(f"Recovery successful for: {self.active\_failsafe.value}")

self.active\_failsafe = None

self.in\_recovery = False

return

# Sleep a bit before checking again

time.sleep(1)

# If we get here, recovery failed within the time limit

logger.error(f"Recovery failed for: {self.active\_failsafe.value}, entering sleep mode")

self.enter\_sleep\_mode()

except Exception as e:

logger.error(f"Error in recovery process: {e}")

self.enter\_sleep\_mode() # Safety measure

finally:

self.in\_recovery = False

def \_attempt\_reconnect\_gnss(self) -> None:

"""Attempt to reconnect to the GNSS receiver"""

logger.info("Attempting to reconnect to GNSS receiver")

if self.simulation\_mode:

# In simulation mode, just wait for the simulated failure to clear

logger.info("Simulation mode: Waiting for GNSS reconnection")

return

# Close existing connection if any

if self.serial\_conn and self.serial\_conn.is\_open:

try:

self.serial\_conn.close()

except Exception as e:

logger.error(f"Error closing serial connection: {e}")

# Try to reconnect

try:

self.serial\_conn = serial.Serial(self.port, self.baud\_rate, timeout=1)

logger.info(f"Reconnected to GNSS receiver on {self.port}")

except Exception as e:

logger.error(f"Failed to reconnect to GNSS receiver: {e}")

def \_attempt\_reconnect\_internet(self) -> None:

"""Attempt to reconnect to the internet"""

logger.info("Attempting to reconnect to internet")

if self.simulation\_mode:

# In simulation mode, just wait for the simulated failure to clear

logger.info("Simulation mode: Waiting for internet reconnection")

return

# In a real implementation, this would attempt to reconnect the cellular modem

# or other internet connection mechanism

def \_attempt\_reconnect\_ntrip(self) -> None:

"""Attempt to reconnect to the NTRIP server"""

logger.info("Attempting to reconnect to NTRIP server")

if self.simulation\_mode:

# In simulation mode, just wait for the simulated failure to clear

logger.info("Simulation mode: Waiting for NTRIP reconnection")

return

# In a real implementation, this would attempt to reconnect to the NTRIP server

def \_attempt\_path\_recovery(self) -> None:

"""Attempt to recover from path deviation"""

logger.info("Attempting to recover from path deviation")

if self.simulation\_mode:

# In simulation mode, just wait for the simulated failure to clear

logger.info("Simulation mode: Adjusting path")

return

# In a real implementation, this would:

# 1. Alert the operator

# 2. Attempt to adjust the path

# 3. If unsuccessful, return to the last waypoint

if self.last\_waypoint:

logger.info(f"Returning to last waypoint: {self.last\_waypoint}")

# Code to navigate back to last waypoint would go here

def enter\_sleep\_mode(self, sleep\_duration: int = None) -> None:

"""

Enter sleep mode for a specified duration (default 5 minutes).

In sleep mode, the rover will stop all operations to prevent dangerous behavior.

"""

if self.in\_sleep\_mode:

logger.warning("Already in sleep mode, ignoring new sleep request")

return

# Use the default sleep duration if none specified

if sleep\_duration is None:

sleep\_duration = self.default\_sleep\_duration

logger.warning(f"Entering sleep mode for {sleep\_duration} seconds")

self.in\_sleep\_mode = True

self.sleep\_start\_time = time.time()

self.sleep\_duration = sleep\_duration

# Signal to safety module if available

if self.safety\_module:

try:

self.safety\_module.emergency\_stop("GNSS Failsafe: " + self.active\_failsafe.value)

except Exception as e:

logger.error(f"Error signaling safety module: {e}")

# Start the sleep thread

sleep\_thread = threading.Thread(target=self.\_sleep\_process, daemon=True)

sleep\_thread.start()

def \_sleep\_process(self) -> None:

"""Process that runs during sleep mode"""

try:

# Sleep for the specified duration

time.sleep(self.sleep\_duration)

# Wake up

self.wake\_up()

except Exception as e:

logger.error(f"Error in sleep process: {e}")

# Try to wake up anyway

self.wake\_up()

def wake\_up(self) -> None:

"""Wake up from sleep mode"""

if not self.in\_sleep\_mode:

logger.warning("Not in sleep mode, ignoring wake up request")

return

logger.info("Waking up from sleep mode")

self.in\_sleep\_mode = False

self.active\_failsafe = None

# Reset all status counters and histories

self.fix\_drops = []

self.position\_history = []

self.drift\_events = []

self.path\_deviation\_events = []

# Signal to safety module if available

if self.safety\_module:

try:

self.safety\_module.clear\_emergency("GNSS Failsafe wake up")

except Exception as e:

logger.error(f"Error signaling safety module: {e}")

# Call the wake up callback if registered

if self.on\_rover\_wakeup:

try:

self.on\_rover\_wakeup()

except Exception as e:

logger.error(f"Error in wake up callback: {e}")

def get\_drift\_severity(self) -> DriftSeverity:

"""

Get the current drift severity level based on drift events

Returns a DriftSeverity enum value

"""

if not self.drift\_events:

return DriftSeverity.NONE

# Calculate average drift size

avg\_drift = sum(d for \_, d in self.drift\_events) / len(self.drift\_events)

# Determine severity based on average drift and number of events

if avg\_drift < 0.05 and len(self.drift\_events) < 3:

return DriftSeverity.LOW

elif avg\_drift < 0.10 and len(self.drift\_events) < 5:

return DriftSeverity.MEDIUM

else:

return DriftSeverity.HIGH

def get\_recommended\_action(self) -> DriftAction:

"""

Get the recommended action based on current GPS status

Returns a DriftAction enum value

"""

# If already in sleep mode, recommend sleep

if self.in\_sleep\_mode:

return DriftAction.SLEEP

# If an active failsafe is being handled, check if recovery is in progress

if self.active\_failsafe:

if self.in\_recovery:

return DriftAction.PAUSE

else:

return DriftAction.STOP

# Otherwise, check drift severity

drift\_severity = self.get\_drift\_severity()

if drift\_severity == DriftSeverity.NONE:

return DriftAction.CONTINUE

elif drift\_severity == DriftSeverity.LOW:

return DriftAction.CONTINUE

elif drift\_severity == DriftSeverity.MEDIUM:

return DriftAction.SLOW\_DOWN

else: # HIGH

return DriftAction.PAUSE

def \_log\_status(self) -> None:

"""Log the current status for monitoring"""

status = {

"timestamp": datetime.now().isoformat(),

"has\_fix": self.has\_fix,

"fix\_type": self.fix\_type,

"satellites": self.satellites,

"pdop": self.pdop,

"hdop": self.hdop,

"position": self.position,

"internet\_connected": self.internet\_connected,

"correction\_age": self.correction\_age,

"ntrip\_data\_rate": self.ntrip\_data\_rate,

"active\_failsafe": self.active\_failsafe.value if self.active\_failsafe else None,

"in\_recovery": self.in\_recovery,

"in\_sleep\_mode": self.in\_sleep\_mode,

"drift\_severity": self.get\_drift\_severity().value,

"recommended\_action": self.get\_recommended\_action().value

}

logger.info(f"GNSS Status: {json.dumps(status, default=str)}")

# In a real implementation, this data might be sent to a monitoring system

def get\_status\_report(self) -> Dict:

"""

Generate a detailed status report for external systems

Returns a dictionary with all relevant status information

"""

current\_time = time.time()

status = {

"timestamp": datetime.now().isoformat(),

"gps": {

"has\_fix": self.has\_fix,

"fix\_type": self.fix\_type,

"satellites": self.satellites,

"pdop": self.pdop,

"hdop": self.hdop,

"vdop": self.vdop,

"position": self.position,

"altitude": self.altitude,

"last\_update": self.last\_gps\_update,

"time\_since\_update": current\_time - self.last\_gps\_update

},

"internet": {

"connected": self.internet\_connected,

"latency": self.internet\_latency,

"last\_check": self.last\_internet\_check,

"time\_since\_check": current\_time - self.last\_internet\_check

},

"corrections": {

"age": self.correction\_age,

"last\_update": self.last\_correction,

"time\_since\_update": current\_time - self.last\_correction,

"ntrip\_data\_rate": self.ntrip\_data\_rate

},

"failsafe": {

"active": self.active\_failsafe.value if self.active\_failsafe else None,

"in\_recovery": self.in\_recovery,

"in\_sleep\_mode": self.in\_sleep\_mode,

"sleep\_start\_time": self.sleep\_start\_time,

"sleep\_duration": self.sleep\_duration,

"time\_in\_sleep": current\_time - self.sleep\_start\_time if self.in\_sleep\_mode else 0

},

"drift": {

"severity": self.get\_drift\_severity().value,

"recommended\_action": self.get\_recommended\_action().value,

"recent\_events": len(self.drift\_events),

"fix\_drops": len(self.fix\_drops)

},

"path": {

"deviation\_events": len(self.path\_deviation\_events),

"has\_planned\_path": len(self.planned\_path) > 0,

"last\_waypoint": self.last\_waypoint

}

}

return status

def check\_path\_deviation(self, current\_position: Tuple[float, float],

current\_heading: float,

planned\_path: List[Tuple[float, float]],

planned\_heading: float) -> Tuple[bool, float, float]:

"""

Check if the rover has deviated from its planned path.

Args:

current\_position: Current (latitude, longitude) position

current\_heading: Current heading in degrees

planned\_path: List of (latitude, longitude) waypoints for the planned path

planned\_heading: Planned heading in degrees

Returns:

Tuple of (deviation\_detected, distance\_deviation, heading\_deviation)

"""

# Find the closest point on the planned path

min\_distance = float('inf')

closest\_index = 0

for i, waypoint in enumerate(planned\_path):

distance = self.\_calculate\_distance(current\_position, waypoint)

if distance < min\_distance:

min\_distance = distance

closest\_index = i

# Calculate the heading of the planned path at this point

if closest\_index < len(planned\_path) - 1:

next\_point = planned\_path[closest\_index + 1]

planned\_segment\_heading = self.\_calculate\_heading(

planned\_path[closest\_index], next\_point)

else:

# Use the provided planned heading if we're at the last waypoint

planned\_segment\_heading = planned\_heading

# Calculate heading deviation (normalize to -180 to 180 degrees)

heading\_diff = current\_heading - planned\_segment\_heading

while heading\_diff > 180:

heading\_diff -= 360

while heading\_diff < -180:

heading\_diff += 360

heading\_deviation = abs(heading\_diff)

# Check against thresholds

distance\_deviation = min\_distance

deviation\_detected = (distance\_deviation > self.thresholds["path\_deviation\_distance"] or # 5 cm threshold

heading\_deviation > self.thresholds["path\_deviation\_heading"]) # 5 degrees threshold

return deviation\_detected, distance\_deviation, heading\_deviation

def \_calculate\_heading(self, point1: Tuple[float, float], point2: Tuple[float, float]) -> float:

"""

Calculate the heading from point1 to point2 in degrees from north.

Args:

point1: Starting position (latitude, longitude)

point2: Ending position (latitude, longitude)

Returns:

Heading in degrees (0-360)

"""

lat1, lon1 = math.radians(point1[0]), math.radians(point1[1])

lat2, lon2 = math.radians(point2[0]), math.radians(point2[1])

# Calculate heading using Great Circle formula

dlon = lon2 - lon1

x = math.sin(dlon) \* math.cos(lat2)

y = math.cos(lat1) \* math.sin(lat2) - math.sin(lat1) \* math.cos(lat2) \* math.cos(dlon)

heading = math.degrees(math.atan2(x, y))

# Normalize to 0-360 degrees

return (heading + 360) % 360

class SimulationExample:

"""

Example usage of the FailsafeModule in simulation mode

"""

def \_\_init\_\_(self):

self.failsafe = FailsafeModule(simulation\_mode=True)

self.failsafe.set\_callbacks(

on\_failsafe\_triggered=self.on\_failsafe,

on\_recovery\_attempt=self.on\_recovery,

on\_rover\_wakeup=self.on\_wakeup

)

def on\_failsafe(self, reason):

print(f"FAILSAFE TRIGGERED: {reason.value}")

def on\_recovery(self, reason):

print(f"RECOVERY STARTED: {reason.value}")

def on\_wakeup(self):

print("ROVER WAKING UP")

def run\_simulation(self, duration=300):

"""Run a simulation for the specified duration in seconds"""

print(f"Starting GNSS failsafe simulation for {duration} seconds")

# Start monitoring

self.failsafe.start\_monitoring()

try:

start\_time = time.time()

while time.time() - start\_time < duration:

# Generate some simulated GNSS data

if not self.failsafe.in\_sleep\_mode:

gnss\_data = self.failsafe.\_generate\_simulated\_gnss\_data()

# Update status based on simulated data

if "fix\_quality" in gnss\_data:

self.failsafe.update\_gps\_status(

has\_fix=(gnss\_data["fix\_quality"] != "NONE"),

fix\_type=gnss\_data["fix\_quality"],

satellites=gnss\_data.get("satellites", self.failsafe.satellites),

hdop=gnss\_data.get("hdop", self.failsafe.hdop),

pdop=gnss\_data.get("pdop", self.failsafe.pdop),

position=gnss\_data.get("position", self.failsafe.position),

altitude=gnss\_data.get("altitude", self.failsafe.altitude)

)

if "correction\_age" in gnss\_data:

self.failsafe.update\_correction\_status(age=gnss\_data["correction\_age"])

if "ntrip\_data\_rate" in gnss\_data:

self.failsafe.update\_ntrip\_data\_rate(data\_rate=gnss\_data["ntrip\_data\_rate"])

# Simulate random internet status

if random.random() < 0.01: # 1% chance to change internet status

self.failsafe.update\_internet\_status(

connected=(random.random() < 0.9), # 90% chance of being connected

latency=random.uniform(50, 200)

)

self.failsafe.update\_module\_communication()

# Print status occasionally

if int(time.time()) % 5 == 0:

status = self.failsafe.get\_status\_report()

action = self.failsafe.get\_recommended\_action()

failsafe\_status = "ACTIVE" if self.failsafe.active\_failsafe else "NORMAL"

print(f"Time: {int(time.time() - start\_time)}s | "

f"Fix: {status['gps']['fix\_type']} | "

f"Sats: {status['gps']['satellites']} | "

f"Status: {failsafe\_status} | "

f"Action: {action.value}")

time.sleep(0.2)

except KeyboardInterrupt:

print("Simulation interrupted")

finally:

# Stop monitoring

self.failsafe.stop\_monitoring()

print("Simulation ended")

if \_\_name\_\_ == "\_\_main\_\_":

"""Run a sample simulation when the module is executed directly"""

sim = SimulationExample()

sim.run\_simulation(duration=120) # Run for 2 minutes

1. logging\_100mm.py

import time

from datetime import datetime

import csv

import os

import random

from threading import Thread, Event

from coordinate\_converter import CoordinateConverter

class GPSLogger:

def \_\_init\_\_(self, rover):

self.rover = rover

self.log\_interval = 0.1 # 100 milliseconds

self.max\_attempts = 3 # Try 3 times before reporting an issue

self.log\_file\_path = r'F:\GPS\task\_2\_waypoints\rover\_log.csv'

self.last\_log\_time = 0

self.running = False

self.stop\_event = Event()

self.logger\_thread = None

self.converter = CoordinateConverter()

# Set run\_id properly

if not hasattr(self.rover, 'run\_id'):

self.rover.run\_id = self.get\_next\_run\_id()

# Ensure log directory exists

os.makedirs(os.path.dirname(self.log\_file\_path), exist\_ok=True)

# Define fieldnames consistently for use throughout the class

self.fieldnames = ['timestamp', 'run\_id', 'x\_utm', 'y\_utm', 'latitude', 'longitude',

'heading', 'bearing', 'compass\_heading', 'fix\_quality',

'satellite\_count', 'deviation', 'data\_age', 'status']

# Write header if file doesn't exist

if not os.path.exists(self.log\_file\_path) or os.path.getsize(self.log\_file\_path) == 0:

with open(self.log\_file\_path, 'w', newline='') as csvfile:

writer = csv.DictWriter(csvfile, fieldnames=self.fieldnames)

writer.writeheader()

def get\_next\_run\_id(self):

"""Determine the next run ID based on existing data in the log file."""

if not os.path.exists(self.log\_file\_path):

return 1

try:

max\_run\_id = 0

with open(self.log\_file\_path, 'r', newline='') as csvfile:

reader = csv.DictReader(csvfile)

for row in reader:

if 'run\_id' in row and row['run\_id']:

try:

run\_id = int(row['run\_id'])

max\_run\_id = max(max\_run\_id, run\_id)

except (ValueError, TypeError):

pass

return max\_run\_id + 1

except Exception as e:

print(f"Error determining run ID: {e}")

return 1

def process\_emlid\_gps\_data(self, emlid\_data):

"""Process real Emlid GPS data (not used in simulation)."""

if not emlid\_data or 'latitude' not in emlid\_data or 'longitude' not in emlid\_data:

print("⚠️ Invalid or missing Emlid GPS data")

return None, None

try:

easting, northing = self.converter.latlon\_to\_utm\_coord(

emlid\_data['latitude'], emlid\_data['longitude']

)

if easting is None or northing is None:

print("⚠️ Failed to convert Emlid lat/lon to UTM")

return None, None

# Calculate local simulation coordinates by applying the offset

utm\_offset\_x = 0

utm\_offset\_y = 0

if hasattr(self.rover, 'navigator') and hasattr(self.rover.navigator, 'utm\_offset\_x'):

utm\_offset\_x = self.rover.navigator.utm\_offset\_x

utm\_offset\_y = self.rover.navigator.utm\_offset\_y

sim\_x = easting - utm\_offset\_x

sim\_y = northing - utm\_offset\_y

self.rover.set\_position(sim\_x, sim\_y)

return easting, northing

except Exception as e:

print(f"Error processing Emlid GPS data: {e}")

return None, None

def get\_gps\_data(self):

"""Get current GPS data from rover, ensuring UTM offsets are applied and valid lat/lon are computed."""

# Get rover heading and calculate bearing

heading = self.rover.heading if hasattr(self.rover, 'heading') else 0.0

bearing = (90 - heading) % 360

compass\_heading = self.rover.get\_compass\_direction(heading) if hasattr(self.rover, 'get\_compass\_direction') else 'Unknown'

# Get rover position

x\_local = self.rover.x if hasattr(self.rover, 'x') else 0.0

y\_local = self.rover.y if hasattr(self.rover, 'y') else 0.0

# Set default UTM offsets (used if navigator is not available)

utm\_offset\_x = 380000.0

utm\_offset\_y = 2044880.0

deviation = 0.0

# Get UTM offsets from navigator if available

if (hasattr(self.rover, 'navigator') and

hasattr(self.rover.navigator, 'utm\_offset\_x') and

self.rover.navigator.utm\_offset\_x is not None and

self.rover.navigator.utm\_offset\_y is not None):

utm\_offset\_x = self.rover.navigator.utm\_offset\_x

utm\_offset\_y = self.rover.navigator.utm\_offset\_y

if hasattr(self.rover.navigator, 'calculate\_deviation'):

deviation = self.rover.navigator.calculate\_deviation(x\_local, y\_local)

# Calculate actual UTM coordinates

x\_utm = x\_local + utm\_offset\_x

y\_utm = y\_local + utm\_offset\_y

# Convert UTM to lat/lon

try:

lat, lon = self.converter.utm\_to\_latlon\_coord(

x\_utm, y\_utm,

zone\_number=45, zone\_letter='N'

)

# Validate lat/lon ranges

if lat is None or lon is None or not (-90 <= lat <= 90) or not (-180 <= lon <= 180):

raise ValueError(f"Invalid lat/lon values: {lat}, {lon}")

fix\_quality = '3D Fix'

satellite\_count = random.randint(8, 12)

status = 'OK'

except Exception as e:

print(f"⚠️ Error converting UTM to Lat/Lon: {e}")

lat, lon = 0.0, 0.0

fix\_quality = 'No Fix'

satellite\_count = 0

status = 'ERROR'

# Create and return the data dictionary

return {

'timestamp': datetime.now().isoformat(),

'run\_id': self.rover.run\_id,

'x\_utm': x\_utm,

'y\_utm': y\_utm,

'latitude': lat,

'longitude': lon,

'heading': heading,

'bearing': bearing,

'compass\_heading': compass\_heading,

'fix\_quality': fix\_quality,

'satellite\_count': satellite\_count,

'deviation': deviation,

'data\_age': 0,

'status': status

}

def log\_data\_once(self):

"""Log a single GPS data entry to the CSV file."""

try:

# Get GPS data

data = self.get\_gps\_data()

# Ensure all required fields have valid values

for field in self.fieldnames:

if field not in data or data[field] is None:

if field in ['x\_utm', 'y\_utm', 'latitude', 'longitude', 'heading',

'bearing', 'deviation', 'data\_age', 'satellite\_count']:

data[field] = 0.0 # Default for numeric fields

elif field == 'status':

data[field] = 'ERROR'

elif field == 'fix\_quality':

data[field] = 'No Fix'

elif field == 'run\_id':

data[field] = self.rover.run\_id

else:

data[field] = ""

# Write to CSV file

with open(self.log\_file\_path, 'a', newline='') as csvfile:

writer = csv.DictWriter(csvfile, fieldnames=self.fieldnames)

writer.writerow(data)

except Exception as e:

print(f"Error in log\_data\_once: {e}")

# Try to write an error record if possible

try:

error\_data = {field: "" for field in self.fieldnames}

error\_data.update({

'timestamp': datetime.now().isoformat(),

'run\_id': getattr(self.rover, 'run\_id', 0),

'status': f'ERROR: {str(e)[:50]}'

})

with open(self.log\_file\_path, 'a', newline='') as csvfile:

writer = csv.DictWriter(csvfile, fieldnames=self.fieldnames)

writer.writerow(error\_data)

except:

pass # If even error logging fails, just continue

def logger\_loop(self):

"""Main logger loop that runs in a separate thread."""

while not self.stop\_event.is\_set():

try:

current\_time = time.time()

time\_since\_last\_log = current\_time - self.last\_log\_time

# If it's time to log data

if time\_since\_last\_log >= self.log\_interval:

self.log\_data\_once()

self.last\_log\_time = current\_time

# Sleep to maintain proper interval

sleep\_time = max(0.01, self.log\_interval - (time.time() - current\_time))

time.sleep(sleep\_time)

except Exception as e:

print(f"Error in logger loop: {e}")

time.sleep(0.1) # Sleep briefly to avoid tight error loops

def start(self):

"""Start the GPS logging thread."""

if self.running:

print("GPS logger is already running")

return

print("📡 Starting GPS data logger (100ms interval)")

self.stop\_event.clear()

self.running = True

self.last\_log\_time = time.time()

self.logger\_thread = Thread(target=self.logger\_loop, daemon=True)

self.logger\_thread.start()

def stop(self):

"""Stop the GPS logging thread."""

if not self.running:

return

print("📡 Stopping GPS data logger")

self.stop\_event.set()

if self.logger\_thread and self.logger\_thread.is\_alive():

self.logger\_thread.join(timeout=1.0)

self.running = False

# Integration functions for the Rover class

def initialize\_gps\_logger(rover):

"""Initialize and start the GPS logger for the rover."""

rover.gps\_logger = GPSLogger(rover)

rover.gps\_logger.start()

return rover.gps\_logger

def stop\_gps\_logger(rover):

"""Stop the GPS logger if it's running."""

if hasattr(rover, 'gps\_logger') and rover.gps\_logger.running:

rover.gps\_logger.stop()

def update\_rover\_position\_from\_emlid(rover, emlid\_data):

"""

Update rover position from Emlid GPS data.

Args:

rover: The rover instance

emlid\_data: Dictionary containing 'latitude' and 'longitude' from Emlid receiver

Returns:

bool: True if position was updated successfully, False otherwise

"""

if not hasattr(rover, 'gps\_logger'):

print("⚠️ GPS logger not initialized")

return False

try:

easting, northing = rover.gps\_logger.process\_emlid\_gps\_data(emlid\_data)

if easting is not None and northing is not None:

return True

return False

except Exception as e:

print(f"Error updating rover position: {e}")

return False

1. emlid\_gps\_integration.py

import serial

import time

import json

import threading

from coordinate\_converter import CoordinateConverter

import os

from serial import SerialException

class EmlidGPSReader:

"""

Class to handle reading GPS data from an Emlid Reach GNSS receiver.

The Emlid receiver outputs NMEA or JSON data which this class parses

and converts to UTM coordinates for the rover navigation system.

"""

def \_\_init\_\_(self, port=None, baud\_rate=115200, message\_format='nmea'):

"""

Initialize the Emlid GPS reader.

Args:

port (str): Serial port where the Emlid receiver is connected

baud\_rate (int): Baud rate for serial communication

message\_format (str): Format of the GPS messages ('json' or 'nmea')

"""

self.port = port

self.baud\_rate = baud\_rate

self.message\_format = message\_format

self.serial\_connection = None

self.reading\_thread = None

self.stop\_thread = threading.Event()

self.converter = CoordinateConverter()

self.last\_position = None

self.last\_update\_time = 0

self.callbacks = []

self.port = port or self.\_autodetect\_emlid\_port()

def \_autodetect\_emlid\_port(self):

# Common ports for Emlid M2 on Windows/Linux

possible\_ports = ['COM3', 'COM8', '/dev/ttyACM0', '/dev/ttyUSB0']

for port in possible\_ports:

if os.path.exists(port):

return port

raise Exception("Emlid M2 not found on common ports!")

def connect(self, retries=3, retry\_delay=2):

"""

Connect to the Emlid receiver with retry logic.

Args:

retries (int): Number of connection attempts (default: 3)

retry\_delay (float): Delay between attempts in seconds (default: 2)

Returns:

bool: True if connection successful, False otherwise

"""

for attempt in range(1, retries + 1):

try:

self.serial\_connection = serial.Serial(

port=self.port,

baudrate=self.baud\_rate,

timeout=1

)

print(f"✅ Connected to Emlid receiver on {self.port} (Attempt {attempt}/{retries})")

return True

except (serial.SerialException, OSError) as e:

retry\_delay \*= 1.5

if attempt < retries:

print(f"⚠️ Connection failed (Attempt {attempt}/{retries}): {e}")

print(f"Retrying in {retry\_delay}s...")

time.sleep(retry\_delay)

else:

print(f"❌ Failed to connect after {retries} attempts")

return False

return True

def send\_rtcm\_data(self, rtcm\_bytes: bytes):

"""

Send raw RTCM correction bytes to the Emlid unit.

"""

if self.serial\_connection and self.serial\_connection.is\_open:

try:

self.serial\_connection.write(rtcm\_bytes)

except Exception as e:

print(f"⚠️ Failed to send RTCM data: {e}")

def disconnect(self):

"""Disconnect from the Emlid receiver."""

if self.serial\_connection and self.serial\_connection.is\_open:

self.serial\_connection.close()

print("✅ Disconnected from Emlid receiver")

def start\_reading(self):

"""Start a thread to continuously read GPS data."""

if not self.serial\_connection or not self.serial\_connection.is\_open:

success = self.connect()

if not success:

return False

self.stop\_thread.clear()

self.reading\_thread = threading.Thread(target=self.\_read\_loop)

self.reading\_thread.daemon = True

self.reading\_thread.start()

print("📡 Started GPS data reading thread")

return True

def stop\_reading(self):

"""Stop the GPS reading thread."""

if self.reading\_thread and self.reading\_thread.is\_alive():

self.stop\_thread.set()

self.reading\_thread.join(timeout=2.0)

print("📡 Stopped GPS data reading thread")

# Disconnect from serial port

self.disconnect()

def register\_callback(self, callback\_func):

"""

Register a callback function to be called when new GPS data is received.

Args:

callback\_func: Function that takes a position dict as parameter

"""

if callback\_func not in self.callbacks:

self.callbacks.append(callback\_func)

def unregister\_callback(self, callback\_func):

"""Remove a callback function from the list."""

if callback\_func in self.callbacks:

self.callbacks.remove(callback\_func)

def \_read\_loop(self):

"""Main loop for reading GPS data from the serial port."""

while not self.stop\_thread.is\_set():

try:

if self.message\_format == 'json':

position = self.\_read\_json()

else:

position = self.\_read\_nmea()

if position:

self.last\_position = position

self.last\_update\_time = time.time()

# Convert lat/lon to UTM

if 'latitude' in position and 'longitude' in position:

easting, northing = self.converter.latlon\_to\_utm\_coord(

position['latitude'], position['longitude']

)

position['easting'] = easting

position['northing'] = northing

# Call all registered callbacks with the position data

for callback in self.callbacks:

try:

callback(position)

except Exception as e:

print(f"Error in GPS callback: {e}")

except Exception as e:

print(f"Error reading GPS data: {e}")

time.sleep(1) # Wait before retrying

def \_read\_json(self):

"""

Read and parse JSON formatted GPS data from Emlid.

Returns:

dict: Parsed GPS position data or None if no valid data

"""

if not self.serial\_connection:

return None

try:

line = self.serial\_connection.readline().decode('utf-8').strip()

if not line:

return None

data = json.loads(line)

# Extract relevant GPS data

if 'position' in data:

position = {

'latitude': data['position'].get('lat', 0),

'longitude': data['position'].get('lon', 0),

'altitude': data['position'].get('height', 0),

'fix\_quality': data.get('solution\_status', 'Unknown'),

'satellites': data.get('satellites\_used', 0),

'hdop': data.get('pdop', 0) # Using PDOP if HDOP not available

}

return position

except json.JSONDecodeError:

pass

except Exception as e:

print(f"Error parsing JSON GPS data: {e}")

return None

def \_read\_nmea(self):

"""Parse NMEA sentences from Emlid M2 and return GPS data dictionary."""

try:

line = self.serial\_connection.readline().decode('utf-8', errors='ignore').strip()

# Parse GPRMC (Recommended Minimum Specific GNSS Data)

if line.startswith('$GPRMC'):

parts = line.split(',')

if len(parts) >= 10 and parts[2] == 'A': # 'A' = Active/Valid fix

lat = self.\_nmea\_to\_decimal(parts[3], parts[4]) # Latitude (DDMM.MMMM, N/S)

lon = self.\_nmea\_to\_decimal(parts[5], parts[6]) # Longitude (DDDMM.MMMM, E/W)

speed = float(parts[7]) if parts[7] else 0.0 # Speed in knots

course = float(parts[8]) if parts[8] else 0.0 # Course in degrees

return {

'latitude': lat,

'longitude': lon,

'speed': speed \* 0.514444, # Convert knots to m/s

'course': course,

'fix\_quality': 'RTK' if 'RTK' in line else 'GPS', # Emlid-specific

'timestamp': parts[1][:6] # HHMMSS

}

# Parse GPGGA (Global Positioning System Fix Data)

elif line.startswith(('$GPGGA', '$GNGGA')):

parts = line.split(',')

if len(parts) >= 15:

return {

'latitude': self.\_nmea\_to\_decimal(parts[2], parts[3]),

'longitude': self.\_nmea\_to\_decimal(parts[4], parts[5]),

'altitude': float(parts[9]) if parts[9] else 0.0,

'satellites': int(parts[7]) if parts[7] else 0,

'hdop': float(parts[8]) if parts[8] else 99.9,

'fix\_quality': {

'0': 'Invalid',

'1': 'GPS',

'2': 'DGPS',

'4': 'RTK Fixed',

'5': 'RTK Float'

}.get(parts[6], 'Unknown')

}

except Exception as e:

print(f"NMEA parsing error: {e}")

return None

def \_nmea\_to\_decimal(self, nmea\_coord, direction):

"""Convert NMEA coordinate (DDMM.MMMM) to decimal degrees."""

if not nmea\_coord or not direction:

return 0.0

try:

degrees = float(nmea\_coord[:2]) if len(nmea\_coord) > 2 else 0.0

minutes = float(nmea\_coord[2:])

decimal = degrees + (minutes / 60.0)

if direction in ('S', 'W'):

decimal \*= -1

return decimal

except ValueError:

return 0.0

def get\_last\_position(self):

"""

Get the last received GPS position.

Returns:

dict: Last GPS position or None if no position has been received

"""

return self.last\_position

def update\_rover\_from\_emlid(rover, emlid\_reader):

"""

Callback function to update rover position from Emlid GPS data.

Args:

rover: The rover instance

emlid\_reader: EmlidGPSReader instance

"""

def on\_gps\_data(position):

if position and 'easting' in position and 'northing' in position:

# Update rover position with UTM coordinates

rover.set\_position(position['easting'], position['northing'])

# Update failsafe module if available

if hasattr(rover, 'failsafe'):

rover.failsafe.update\_gps\_status(

has\_fix = position['fix\_quality'] in ['GPS', 'DGPS', 'RTK Fixed', 'RTK Float'],

satellites=position['satellites'],

hdop=position['hdop']

)

# Register the callback with the Emlid reader

emlid\_reader.register\_callback(on\_gps\_data)

def setup\_emlid\_integration(rover):

"""

Set up Emlid GPS integration with the rover.

Args:

rover: The rover instance

Returns:

EmlidGPSReader: Configured GPS reader instance

"""

# Initialize the GPS reader

emlid\_reader = EmlidGPSReader()

# Configure rover to use Emlid GPS data

update\_rover\_from\_emlid(rover, emlid\_reader)

# Initialize the GPS reader

gps\_reader = EmlidGPSReader()

# Try connecting with 5 retries and 3-second delays

success = gps\_reader.connect(retries=5, retry\_delay=3)

if success:

print("Connected! Starting data collection...")

else:

print("Failed to connect. Check hardware and try again.")

# Start reading GPS data

success = emlid\_reader.start\_reading()

if not success:

print("⚠️ Failed to start Emlid GPS reader")

return emlid\_reader

1. coordinate\_converter.py

import utm

import math

class CoordinateConverter:

"""

Handles conversion between WGS84 lat/lon coordinates (EPSG:4326) and

UTM coordinates (EPSG:32645 - UTM zone 45N) for the rover navigation system.

"""

def \_\_init\_\_(self):

# Default UTM zone for EPSG:32645 is zone 45N

self.default\_zone = 45

self.default\_zone\_letter = 'N'

self.zone\_number = None

self.zone\_letter = None

def latlon\_to\_utm\_coord(self, lat, lon):

"""

Convert latitude/longitude (EPSG:4326) to UTM coordinates (EPSG:32645).

Args:

lat (float): Latitude in decimal degrees

lon (float): Longitude in decimal degrees

Returns:

tuple: (easting, northing, zone\_number, zone\_letter)

"""

try:

easting, northing, zone\_number, zone\_letter = utm.from\_latlon(lat, lon)

self.zone\_number = zone\_number

self.zone\_letter = zone\_letter

return easting, northing, zone\_number, zone\_letter

except Exception as e:

print(f"Error converting lat/lon to UTM: {e}")

return None, None, None, None

def utm\_to\_latlon\_coord(self, easting, northing, zone\_number=None, zone\_letter=None):

"""

Convert UTM coordinates (EPSG:32645) to latitude/longitude (EPSG:4326).

Args:

easting (float): UTM easting coordinate in meters

northing (float): UTM northing coordinate in meters

zone\_number (int, optional): UTM zone number. Defaults to 45.

zone\_letter (str, optional): UTM zone letter. Defaults to 'N'.

Returns:

tuple: (latitude, longitude) in decimal degrees

"""

try:

zone\_number = zone\_number or self.zone\_number or self.default\_zone

zone\_letter = zone\_letter or self.zone\_letter or self.default\_zone\_letter

lat, lon = utm.to\_latlon(easting, northing, zone\_number, zone\_letter)

return lat, lon

except Exception as e:

print(f"Error converting UTM to lat/lon: {e}")

return None, None

def get\_distance\_between\_coords(self, lat1, lon1, lat2, lon2):

R = 6371000 # Radius of Earth in meters

lat1\_rad = math.radians(lat1)

lon1\_rad = math.radians(lon1)

lat2\_rad = math.radians(lat2)

lon2\_rad = math.radians(lon2)

dlat = lat2\_rad - lat1\_rad

dlon = lon2\_rad - lon1\_rad

a = math.sin(dlat/2)\*\*2 + math.cos(lat1\_rad) \* math.cos(lat2\_rad) \* math.sin(dlon/2)\*\*2

c = 2 \* math.atan2(math.sqrt(a), math.sqrt(1 - a))

return R \* c

def get\_bearing\_between\_coords(self, lat1, lon1, lat2, lon2):

lat1\_rad = math.radians(lat1)

lon1\_rad = math.radians(lon1)

lat2\_rad = math.radians(lat2)

lon2\_rad = math.radians(lon2)

y = math.sin(lon2\_rad - lon1\_rad) \* math.cos(lat2\_rad)

x = math.cos(lat1\_rad) \* math.sin(lat2\_rad) - \

math.sin(lat1\_rad) \* math.cos(lat2\_rad) \* math.cos(lon2\_rad - lon1\_rad)

bearing = math.degrees(math.atan2(y, x))

return (bearing + 360) % 360

1. Ntrip\_client

# ntrip\_client.py

import requests

from requests.auth import HTTPBasicAuth

class NTRIPClient:

def \_\_init\_\_(self, host, port, mountpoint, user, password):

self.url = f"http://{host}:{port}/{mountpoint}"

self.auth = HTTPBasicAuth(user, password)

self.session = requests.Session()

self.headers = {

'Ntrip-Version': 'Ntrip/2.0',

'User-Agent': 'NTRIP EmlidClient/1.0',

'Connection': 'close',

'Accept': '\*/\*'

}

def get\_corrections(self):

"""Stream RTCM correction data from NTRIP caster."""

try:

response = self.session.get(

self.url,

auth=self.auth,

stream=True,

headers=self.headers,

timeout=10

)

response.raise\_for\_status()

print("✅ Connected to NTRIP caster")

for chunk in response.iter\_content(chunk\_size=1024):

if chunk:

yield chunk # RTCM data bytes

except Exception as e:

print(f"❌ NTRIP error: {e}")